



**Create a code snippet app**

*by Michelangelo Altamore*

**Open Source Rails project**

*by Dmitry Amelchenko*

**Theme Support**

*by James Stewart*

**Observer and Singleton**

**design patterns in Ruby**

*by Khaled al Habache*

**JRuby monitoring with JMX**

*by Joshua Moore*

**Ruby Web Frameworks:**

**A Dive into Waves**

*by Carlo Pecchia*

**How to Implement**

**Automated Testing**

*by Eric Anderson*

**Ruby on Rails & Flex**

*by Arturo Fernandez*

**Workflow solutions with AASM**

*by Chee Yeo*

**MeshU Exclusive Coverage**

**How to build a business with Open Source**

*by Chris Wanstrath*

**Carl Mercier \* Ilya Grigorik \* Ryan Singer**

*interviews by Rupak Ganguly*

# **Rails Magazine**

**fine articles on Ruby & Rails**

RPM gives your developers **transparency** into the performance details of our applications, **saving significant time** in isolating bottlenecks.

New Relic's RPM has **dramatically improved** the way we monitor the health and performance of our applications.

If you are running a Rails app of any reasonable size **you *have to use* New Relic.**

— Tobias Luetke, Shopify

With RPM we had **more time to add new features** and fix issues, rather than scanning through piles of log files.

We used the other tools out there, and were **not happy until we switched to New Relic**. I was amazed how much we found in just a few hours of monitoring. We got the **real-time picture we had been dreaming about**.

New Relic RPM lets you see and understand performance metrics in real time so you can fix problems fast. It's intuitive. It's granular. And, it's a 10-second Rails plug-in install.

Take **RPM Lite** for a spin — it's free!



**New Relic**  
Rails Performance Management



# A Word from the Editor

## by Olimpiu Metiu

Welcome to the largest edition of Rails Magazine yet!

The past couple of months were hectic as usual, with our team expanding to 7 people (<http://railsmagazine.com/team>), not including our many authors and columnists (<http://railsmagazine.com/authors>). Should you find an article particularly useful or enjoyable, please take a moment to send a thank you note to the appropriate author or editor – they worked hard on a volunteer basis to share their knowledge in a professional format.

To help the magazine continue on, we are looking for additional help – please contact us if interested in joining the editorial team.

**Olimpiu Metiu** is a Toronto-based architect and web strategist.

He led the Emergent Technologies group at Bell Canada for the past couple of years, and his work includes many of Canada's largest web sites and intranet portals.

Recently, Olimpiu accepted a position with Research in Motion (the maker of BlackBerry), where he is responsible with the overall architecture of an amazing collaboration platform.



A long-time Rails enthusiast, he founded Rails Magazine as a way to give back to this amazing community.

Follow on **Twitter**: <http://twitter.com/olimpiu>  
Connect on LinkedIn: <http://www.linkedin.com/in/ometiu>

Our RailsConf 2009 edition, intended originally as a 2-page event report, took a life of its own and ended up with 12 pages of exclusive event coverage, Rails 3 information and interviews with prominent rubyists.

Of course, these are exciting times for the entire Rails community: Passenger for Nginx, MacRuby/HotCocoa, Ruby 1.9 gem compatibility and Rails 3 rumours are just a few things keeping everyone busy. Also, the Matt-gate incident led to self-reflection and positive action, including the creation of [Railsbridge.org](http://Railsbridge.org) – an inclusive community we hope will add value to the community and complement Rails Activism efforts.

Without further ado, Rails Magazine Issue #3...

DISCUSS: <http://railsmagazine.com/3/1>

## Contents

Editorial .....	3
<i>by Olimpiu Metiu</i>	
Create a simple code snippet app with Rails.....	4
<i>by Michelangelo Altamore</i>	
Working on a typical Open Source Rails project .....	12
<i>by Dmitry Amelchenko</i>	
Theme Support .....	16
<i>by James Stewart</i>	
Observer and Singleton design patterns in Ruby.....	17
<i>by Khaled al Habache</i>	
JRuby Tip: Monitoring with JMX.....	20
<i>by Joshua Moore</i>	
Workflow solutions using AASM.....	22
<i>by Chee Yeo</i>	
Ruby Web Frameworks: A Dive into Waves .....	26
<i>by Carlo Pecchia</i>	
How to Implement Automated Testing .....	32
<i>by Eric Anderson</i>	
Ruby on Rails & Flex:	
Building a new software generation.....	35
<i>by Arturo Fernandez</i>	
Building a Business with Open Source.....	42
<i>by Chris Wanstrath</i>	
Interview with Carl Mercier.....	46
Interview with Ilya Grigorik .....	48
Interview with Ryan Singer.....	50
<i>(interviews by Rupak Ganguly)</i>	

# Create a simple code snippet app with Rails

by Michelangelo Altamore

In this article you will see how to create a basic source code snippet application. I will use tests to drive development step by step and provide a practical example.

I assume that you are familiar with Ruby on Rails basic concepts. You should have a working environment with a recent version of Ruby ( $\geq 1.8.7$ ), RubyGems ( $\geq 1.3.0$ ) and, of course, Rails ( $\geq 2.1$ ).

## Initial Sketching

We create a new rails application called Snippetty:

```
$ rails snippetty; cd snippetty
```

and install `nifty_generators` by Ryan Bates as a plugin by running:

```
$ script/plugin install git://github.com/ryanb/nifty-generators.git
```

so that we can quickly generate a simple layout with the command:

```
$ script/generate nifty_layout
```

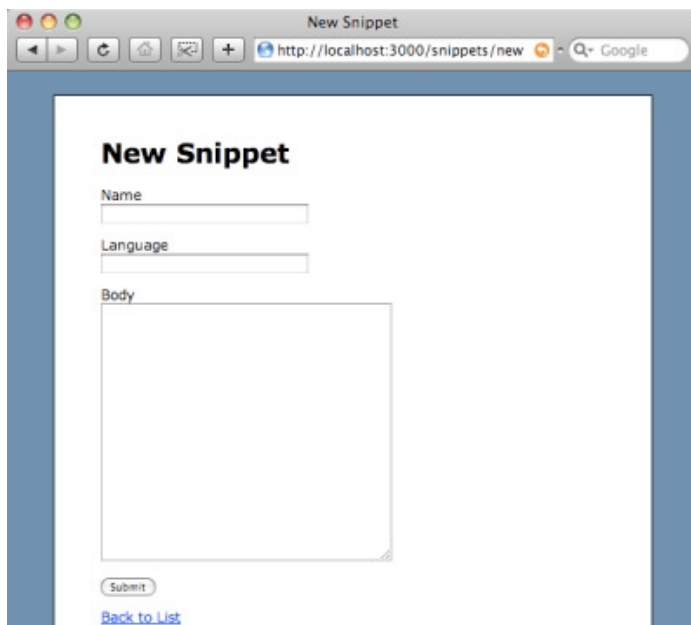
A code snippet will have the following attributes:

- name
- language
- body

Let's scaffold a basic model:

```
$ script/generate nifty_scaffold snippet name:string language:string body:text
```

Migrate the database with `rake db:migrate`, start your web server with `script/server` and point your browser to `localhost:3000/snippets` to see a functional snippet view.



Soon after playing with the web interface we realize that our fresh Snippetty is lacking a few things:

1. language is in a free text field instead of a selectable list of available languages
2. snippet name, language and body are not required neither on create nor on update
3. snippet body is not highlighted according to the snippet language
4. two different snippets can have the same name
5. the list of snippet show a tabular list instead of a list of snippets

These functionalities represent Snippetty's business value. We can start by constraining a snippet to have a unique name, a language and a body; after that we will try to get syntax highlighting.

In general you should not care too much about the order of activities when the sequence is not critical, and this is the case since we can't ship a unit of work missing either syntax highlighting or with a potentially inconsistent snippet collection.

Before start, I rearrange our previous list according to their priority and express each as an expectation:

1. each snippet should have a mandatory unique name, language and body
2. each snippet body should be rendered highlighted according to its language
3. index action should list snippets in a blog-style way
4. snippet language should be chosen from a selectable list of languages instead of free text

Now that we know where to head for, we can start our first iteration.

## Task 1: Snippet Validation

The business logic of Snippet requires a mandatory unique name, a stated programming language and a body. We will implement this starting with tests.

Test Driven Development (TDD) basically means that for any new requirement we first add a new test asserting what should happen for a case and then we will implement code so that the new test, and all of the old tests, are satisfied.

Let's start by editing `test/unit/snippet_test.rb` and implementing a test method named `test_should_have_a_name` that fails when the snippet's name is not present:

```
1 require 'test_helper'
2
3 class SnippetTest < ActiveSupport::TestCase
```

```

4   def test_should_have_a_name
5     snippet = Snippet.new
6     assert_nil snippet.name
7     snippet.valid?
8     assert_not_nil snippet.errors.on(:name)
9   end
10 end

```

Take a break to understand what this test is saying. On line 5 we are instantiating a new snippet. On the next line we assert that our snippet has no default name, that is `snippet.name` should evaluate to `nil`. Then we run the active record validation on line 7 by sending the `valid?` message on the snippet object.

We have fulfilled our preconditions, so on line 8 we assert that we should have an error for the snippet object on the `name` symbol.

That is sufficient to express that a snippet validation should fail when it has no name. Run `rake test:units` on your console to see the following failure:

```

1) Failure:
test_should_have_a_name(SnippetTest)
[./test/unit/snippet_test.rb:8:in `test_should_have_a_name']
<nil> expected to not be nil.
1 tests, 2 assertions, 1 failures, 0 errors

```

The next step is to make the test pass by implementing the simplest thing that could possibly work. Let's modify `app/models/snippet.rb` so that it looks like that:

```

class Snippet < ActiveRecord::Base
  validates_presence_of :name
end

```

and run `rake test:units` again.

```

Started
.
Finished in 0.076376 seconds.
1 tests, 2 assertions, 0 failures, 0 errors

```

The test passes! We are done with that iteration. Now it's your turn: using the above code as a guide try to write a test that fails when the `language` attribute is not set on a snippet object. Then implement the simplest thing that makes your test pass. Do the same for the `body` attribute and finally confront your snippet unit test suite with mine.

To accomplish task 1 we still need a unique name attribute for our snippets. Consider the following test method:

```

25 def test_should_have_a_unique_name
26   snippet_one = Snippet.create(:name => 'Hello
World', :language => "ruby", :body => "puts \"Hello

```

```

World\"")
27   assert_nil snippet_one.errors.on(:name)
28   snippet_two = Snippet.create(:name => snippet_one.name, :language => "ruby", :body => "def hello;
#{snippet_one.body}; end")
29   assert_not_nil snippet_two.errors.on(:name)
30 end

```

We instantiate two snippet objects, we assert that the first one is created and saved without errors on name in line 27, while the second one on line 28 is expected to have an error on name, having the same of the first. Run the usual `rake test:units` to see:

```

Started
...F
Finished in 0.099406 seconds.
1) Failure:
test_should_have_a_unique_name(SnippetTest)
[./test/unit/snippet_test.rb:33:in `test_should_have_a_unique_name']
<nil> expected to not be nil.
4 tests, 8 assertions, 1 failures, 0 errors

```

As you may have guessed we need to change `app/models/snippet.rb` like that:

```

class Snippet < ActiveRecord::Base
  validates_presence_of :name, :language, :body
  validates_uniqueness_of :name
end

```

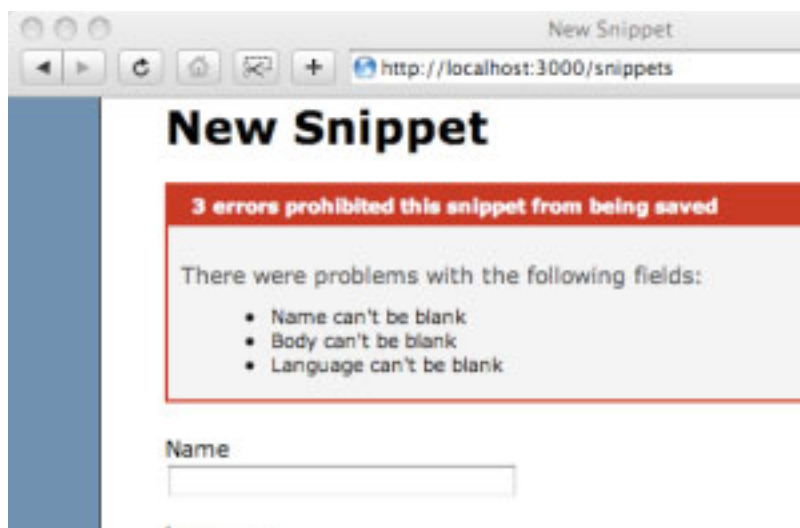
Let's try our tests again:

```

Started
....
Finished in 0.095361 seconds.
4 tests, 8 assertions, 0 failures, 0 errors

```

All tests are passing now. We have validation for our snippets and the first task is complete.



Time to move on the next one.

## Task 2: Snippet Highlighting

For this task we will use the library CodeRay by Kornelius Kalnbach. Check if it is already installed on your system with `gem list coderay`. If it is not listed you can install by running:

```
$ sudo gem install coderay.
```

Our aim is to let Snippet being able to use the `coderay` gem to render an highlighted version of a snippet body according to its language syntax.

So we configure that dependency in `config/environment.rb` by adding the line:

```
config.gem "coderay"
```

We still don't know how to interact with that library. However that knowledge is not that far away, by running `ri CodeRay` you can see its usage; I report here what is relevant for us:

Highlight Ruby code in a string as html

```
require 'coderay'
print CodeRay.scan('puts "Hello, world!"', :ruby).html
```

# prints something like this:

```
puts <span class="s">&quot;Hello, world!&quot;</span>
```

Highlight C code from a file in a html div

```
require 'coderay'
print CodeRay.scan(File.read('ruby.h'), :c).div
print CodeRay.scan_file('ruby.h').html.div
```

You can include this div in your page. The used CSS styles can be

printed with

```
% coderay_stylesheet
```

```
...
```

The documentation tells us that:

1. we can obtain `coderay`'s stylesheet with the command `coderay_stylesheet`
2. we can highlight a string calling `scan` method on the `CodeRay` class

In order to integrate `coderay` stylesheet we run the following inside root folder of Snippet:

```
$ coderay_stylesheet > public/stylesheets/coderay.css
```

You should obtain the stylesheet file. To get it loaded we must modify application layout, as you can see on line 6 in `app/views/layouts/application.html.erb` file:

```
...
4 <head>
5 <title><%= h(yield(:title) || "Untitled") %></
title>
6 <%= stylesheet_link_tag 'coderay', 'application'
%>
7 <%= yield(:head) %>
8 </head>
...
```

The stylesheet should be loaded now (you can look at the source to be sure).

```
<html>
<head>
  <title>Snippets</title>
  <link href="/stylesheets/coderay.css?1233480447" media="screen" r
  <link href="/stylesheets/application.css?1233422840" media="screen" r
</head>
<body>
```

Now it's time to explore `CodeRay` from the Rails console:

```
$ script/console
```

```
Loading development environment (Rails 2.2.2)
```

*A bridge too red*



```
>> CodeRay
=> CodeRay
```

The gem has been correctly loaded if you can see that. Let's try now to get syntax highlighting using the same example as the user manual:

```
>> CodeRay.scan('puts "Hello, world!"', :ruby).html
=> "puts <span class=\"s\"><span class=\"dl\">&quot;</span><span class=\"k\">Hello, world!</span><span class=\"dl\">&quot;</span></span>"
```

We see a bunch of `span` tags with their own CSS class, but we would need a `div` mentioning a `CodeRay` CSS class. Let's try again calling the `div` method on it:

```
>> CodeRay.scan('puts "Hello, world!"', :ruby).html.div
=> "<div class=\"CodeRay\">\n <div class=\"code\"><pre>puts <span class=\"s\"><span class=\"dl\">&quot;</span><span class=\"k\">Hello, world!</span><span class=\"dl\">&quot;</span></span></pre></div>\n</div>\n"
```

It looks much better now. It has a `div` with a `CodeRay` class and the code is inside a `pre` tag so that multiline code will be shown on separate lines.

We now have enough ingredients for the following test:

```
def test_should_render_highlighted_html
  plain_body = %Q(puts "Hello, world!")
  highlighted_body = %Q(<div class=\"CodeRay\">\n <div class=\"code\"><pre>puts <span class=\"s\"><span class=\"dl\">&quot;</span><span class=\"k\">Hello, world!</span><span class=\"dl\">&quot;</span></span></pre></div>\n</div>\n)

  snippet = Snippet.new(:name => "Hello", :language => "ruby", :body => plain_body)
  assert_equal highlighted_body, snippet.highlight
end
```

First we instantiate a Ruby snippet with the content of `puts "Hello, world!"` as body and with the requirement that it should be rendered by the same markup that we last saw in the console. We run our unit test suite as usual and we get:

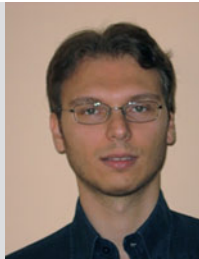
```
Started
...E
Finished in 0.098842 seconds.
1) Error:
test_should_render_highlighted_html(SnippetTest):
NoMethodError: undefined method 'highlight' for #<Snippet:0x211b460>
./test/unit/snippet_test.rb:44:in 'test_should_render_highlighted_html'
5 tests, 8 assertions, 0 failures, 1 errors
```

It complains since we do not still have any highlight

method. So we add it to `app/models/snippet.rb`:

```
def highlight
end
```

**Michelangelo Altamore** is an Italian Ruby on Rails evangelist with a passion for expressive beautiful code. He holds a B.S. in Computer Science from Catania University, Italy. Michelangelo has more than 3 years of experience in software development and offers his services from Convergent.it (<http://convergent.it>)



Rerun the test to find that now we've got a different problem:

```
Started
...F
Finished in 0.102812 seconds.
1) Failure:
test_should_render_highlighted_html(SnippetTest)
./test/unit/snippet_test.rb:44:in 'test_should_render_highlighted_html'
<"<div class=\"CodeRay\">\n <div class=\"code\"><pre>puts <span class=\"s\"><span class=\"dl\">&quot;</span><span class=\"k\">Hello, world!</span><span class=\"dl\">&quot;</span></span></pre></div>\n</div>\n">
  expected but was
<nil>
5 tests, 9 assertions, 1 failures, 0 errors
```

It fails since the `highlight` method actually returns `nil`. We are ready to implement source highlighting by writing the implementation that we have already seen using the console and hopefully making the tests pass:

```
def highlight
  CodeRay.scan(self.body, self.language).html.div
end
```

We try our tests again:

```
Started
...
Finished in 0.111017 seconds.
5 tests, 9 assertions, 0 failures, 0 errors
```

And they pass! We can highlight source code snippets now and we have a test that confirms that. However, we can't show highlight source code for anyone until we modify the snippet views.

Your first instinct could be to look for snippet views,



manually customize them and finish the work.

That would be great, however you will end up without any tests for your controller or views, and that is not good. We instead maintain discipline and proceed with the next task.

## Task 3: Action Views Customization

Let's explore functional tests with rake test:functionals:

```
Started
.....
Finished in 0.259097 seconds.
9 tests, 10 assertions, 0 failures, 0 errors
```

You may wonder how it is possible that you already have a suite of 9 different passing functional tests without even writing a single one.

You should thank Ryan for this, he is so good that `nifty_` generators comes with functional tests not only for `Test::Unit` but also for `Shoulda` and `RSpec`.

That simply means that our work for functional tests will be less than expected. So open `test/functionals/snippets_controller_test.rb` with your editor and have a look at the first method:

```
1 require 'test_helper'
2
3 class SnippetsControllerTest <
  ActionController::TestCase
4   def test_index
5     get :index
6     assert_template 'index'
7   end
  ...
end
```

As the name suggests, the method is testing the index action of the snippets controller. On line 5, there is a call to an HTTP request, in particular the `get` method; the symbol `index`, that actually stands for the index action of our snippets view, is passed as an argument to the `get` request. That request is expected to produce a response rendering the index template view for the snippet controller. This is fine and it works, we just would like to add the expectation that a list of snippets is rendered on the template. To do that we modify the method in that way:

```
4 def test_index
5   get :index
6   assert_template 'index'
7   snippets = assigns(:snippets)
8   assert_select 'div#snippets' do
9     assert_select 'div.CodeRay', :count => snippets.
size
10  end
```

...

On line 7 we are assigning to snippets the fixtures set contained in `snippets.yml` that you can see under the `test/fixtures` directory, and we expect that our template contains a snippets id div tag, and inside it, a number of div with `CodeRay` class matching the number of the snippets.

Running our functional tests we see:

```
Started
...F...
Finished in 0.227583 seconds.
1) Failure:
test_index(SnippetsControllerTest)
Expected at least 1 element matching "div#snippets",
found 0.
<false> is not true.
9 tests, 11 assertions, 1 failures, 0 errors
```

Our test is failing. Indeed, we have no `div#snippets` for our view. We can implement that and produce the `div.CodeRay` listing with the following `index.html.erb`:

```
<% title "Snippet" %>
<h2><%= link_to "Create a new code snippet", new_snippet_path %></h2>
<hr/>

<h2>View available code snippets</h2>
<div id="snippets">
  <% for snippet in @snippets %>
    <h3>
      <%=h snippet.name %> – <%=h snippet.language %>
    </h3>
    <small>
      <%= link_to "Show", snippet %> |
      <%= link_to "Edit", edit_snippet_path(snippet) %> |
      <%= link_to "Destroy", snippet, :confirm => 'Are you sure?', :method => :delete %>
    </small>
    <%= snippet.highlight %>
  <hr/>
  <% end %>
</div>
```

Check our functional tests:

```
Started
.....
Finished in 0.22947 seconds.
9 tests, 13 assertions, 0 failures, 0 errors
```

And they pass. In fact, you can now see `div#snippets` and `div.CodeRay` by looking at the source code of the index page.





By the way, we've just finished our 3<sup>rd</sup> requirement. Now, let's try to modify our show action to properly display a snippet. We add line 16 in `snippets_controller_test.rb`:

```
13 def test_show
14   get :show, :id => Snippet.first
15   assert_template 'show'
16   assert_select 'div.CodeRay', :count => 1
17 end
```

Here we expect a `div.CodeRay` element on our page. The test fails since our generated show view action does not know anything about syntax highlighting, as you can see:

Started

.....F..

Finished in 0.306102 seconds.

1) Failure:

test\_show(SnippetsControllerTest)

Expected at most 1 element matching "div.CodeRay", found 0.

<false> is not true.

9 tests, 14 assertions, 1 failures, 0 errors

Now we produce the following template for `show.html.erb`:

```
<% title "#{snippet</span>.name<span class="id">}</span></span><span class="k"> - </span><span class="il"><span class="id">#{</span><span class="iv">snippet.language}" %>
<small>
  <%= link_to "Edit", edit_snippet_path(snippet</span>)
<span class="id">%&gt;</span></span> |
  <span class="il"><span class="id">&lt;%</span> link_to <span class="s"><span class="dl">&quot;</span><span class="k">Destroy</span><span class="dl">&quot;</span></span></span>, <span class="iv">snippet, :confirm => 'Are you sure?', :method => :delete %> |
  <%= link_to "View All", snippets_path %>
</small>
<%= @snippet.highlight %>
```

Finally, our tests are now happily passing:

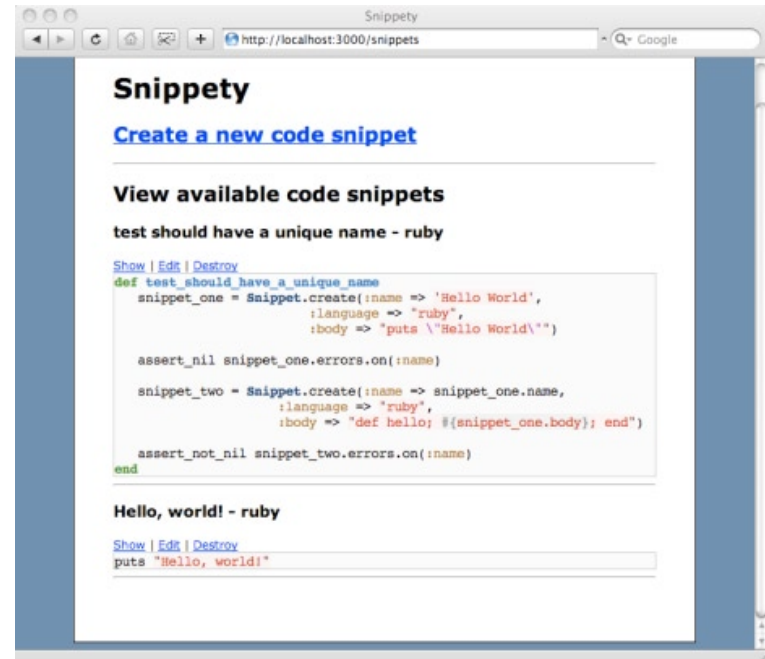
Started

.....

Finished in 0.271848 seconds.

9 tests, 15 assertions, 0 failures, 0 errors

We can take a break now and have a look at our application's front end. After the creation of a couple of code snippets, Snippetty now looks like this:



While editing or creating a code snippet we have no selectable list of available languages. It's time to address the issue.

We start by adding the requirement that a language should be presented inside a select box by placing that assertion on line 22 of `snippets_controller_test.rb`:

```
19 def test_new
20   get :new
21   assert_template 'new'
22   assert_select 'select#snippet_language'
23 end
```

The above expresses that a template should contain a select element with `snippet_language` id and a parameter corresponding to the language attribute in the snippet model.

Having no select box in our view, rake test:functionals fails as shown:

Started

.....F...

Finished in 0.282672 seconds.

1) Failure:

```
test_new(SnippetsControllerTest)
Expected at least 1 element matching "select#snippet_
language", found 0.
<false> is not true.
9 tests, 16 assertions, 1 failures, 0 errors
```

We find that the relevant code to modify is placed inside `_form.html.erb`, a view partial:

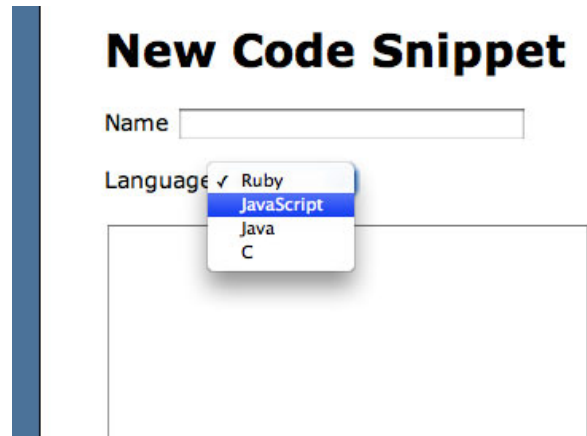
```
1 <% form_for @snippet do |f| %>
2   <%= f.error_messages %>
3   <p>
4     <%= f.label :name %><br />
5     <%= f.text_field :name %>
6   </p>
7   <p>
8     <%= f.label :language %><br />
9     <%= f.text_field :language %>
10  </p>
11  <p>
12    <%= f.label :body %><br />
13    <%= f.text_area :body %>
14  </p>
15  <p><%= f.submit "Submit" %></p>
16 <% end %>
```

We change creating a local variable holding an array of languages on line 8 and changing the `text_field` into a select box on line 11, as follows:

```
8 <% form_for snippet</span> <span class="r">do</
span> |f| <span class="idl">%&gt;</span></span>
<span class="no"> 2</span> <span class="il"><span
class="idl">&lt;%=</span> f.error_messages <span
class="idl">%&gt;</span></span>
<span class="no"> 3</span> <span class="ta">&lt;p&gt;</
span>
<span class="no"> 4</span> <span class="il"><span
class="idl">&lt;%=</span> f.label <span class="sy">:name</
span> <span class="idl">%&gt;</span></span><span
class="ta">&lt;br</span> <span class="ta">/&gt;</span>
<span class="no"> 5</span> <span class="il"><span
class="idl">&lt;%=</span> f.text_field <span
class="sy">:name</span> <span class="idl">%&gt;</span></
span>
<span class="no"> 6</span> <span class="ta">&lt;/
p&gt;</span>
<span class="no"> 7</span> <span class="ta">&lt;p&gt;</
span>
<span class="no"> 8</span> <span class="il"><span
class="idl">&lt;%=</span> <span class="iv">languages = [
"Ruby","JavaScript","Java","C"] %>
9 <p>
10 <%= f.label :language %>
```

```
11 <%= f.select :language, @languages %>
12 </p>
13 <p>
14   <%= f.label :body %><br />
15   <%= f.text_area :body %>
16 </p>
17 <p><%= f.submit "Submit" %></p>
18 <% end %>
```

```
rake test:functionals says that:
Started
.....
Finished in 0.3083 seconds.
```



```
9 tests, 16 assertions, 0 failures, 0 errors
```

We have no failures. It means that we have a working select list:

We have caused some eyebrow raising since a good candidate model property is placed inside a local variable of the view layer.

We should refactor it and refactor it without making the tests fail. You can see how in `snippet.rb`:

```
1 class Snippet < ActiveRecord::Base
2   LANGUAGES = %w(Ruby JavaScript Java C)
3   ...
4 end
```

And in `_form.html.erb`:

```
7 <p>
8   <%= f.label :language %>
9   <%= f.select :language, Snippet::LANGUAGES %>
10 </p>
```

```
Sure enough, we see that the tests are still passing:
Started
.....
```

Finished in 0.43375 seconds.

9 tests, 16 assertions, 0 failures, 0 errors

As a side effect, the form partial change also provided a working implementation for the edit action, without its own test. As a simple exercise you can modify the `test_edit` methods on your functional test suite for covering that action. If you are tired of using rake to launch your test suite, you can take a look at `autotest`.

## Conclusion

Initial requirements for `snippety` have been fulfilled. If it this exercise were real, it would be now released to the users and we would await feedback from them. Maybe they would like more languages, the addition of line numbers, etc.

What is important is that now you should be more familiar with TDD. As you have seen, the concept is easy to grasp and you can use it effectively on your own projects.

Indeed `Test::Unit` is great but some people are happier when they can minimize Ruby's syntactic sugar impact on tests, still using it to express their tests in a readable language that is meaningful for the application. `Shoulda` by `Thoughtbot` addresses this issue. You can learn by their nice tutorial.

I can't close without mentioning `RSpec`, the original Behaviour Driven Development framework for Ruby. If you feel

curious, you could have a look at a great talk by Dave Astels and David Chelimsky where it is explained what BDD is, and where it comes from. By the way, if you are interested, `The RSpec Book` (beta) has been recently published.

I hope you've find it useful and thank you for your time.

## Resources

<http://github.com/ryanb/nifty-generators/tree/master>

<http://github.com/ryanb>

<http://coderay.rubychan.de/>

<http://railscasts.com/>

<http://github.com/thoughtbot/shoulda/tree/master>

<http://github.com/dchelimsky/rspec/tree/master>

[http://ph7spot.com/articles/getting\\_started\\_with\\_autotest](http://ph7spot.com/articles/getting_started_with_autotest)

<http://thoughtbot.com/>

[http://rubyconf2007.confreaks.com/d3t1p2\\_rspec.html](http://rubyconf2007.confreaks.com/d3t1p2_rspec.html)

<http://blog.daveastels.com/>

<http://blog.davidchelimsky.net/>

<http://www.pragprog.com/titles/achbd/the-rspec-book>

DISCUSS: <http://railsmagazine.com/3/2>

*Looking Up from Below*



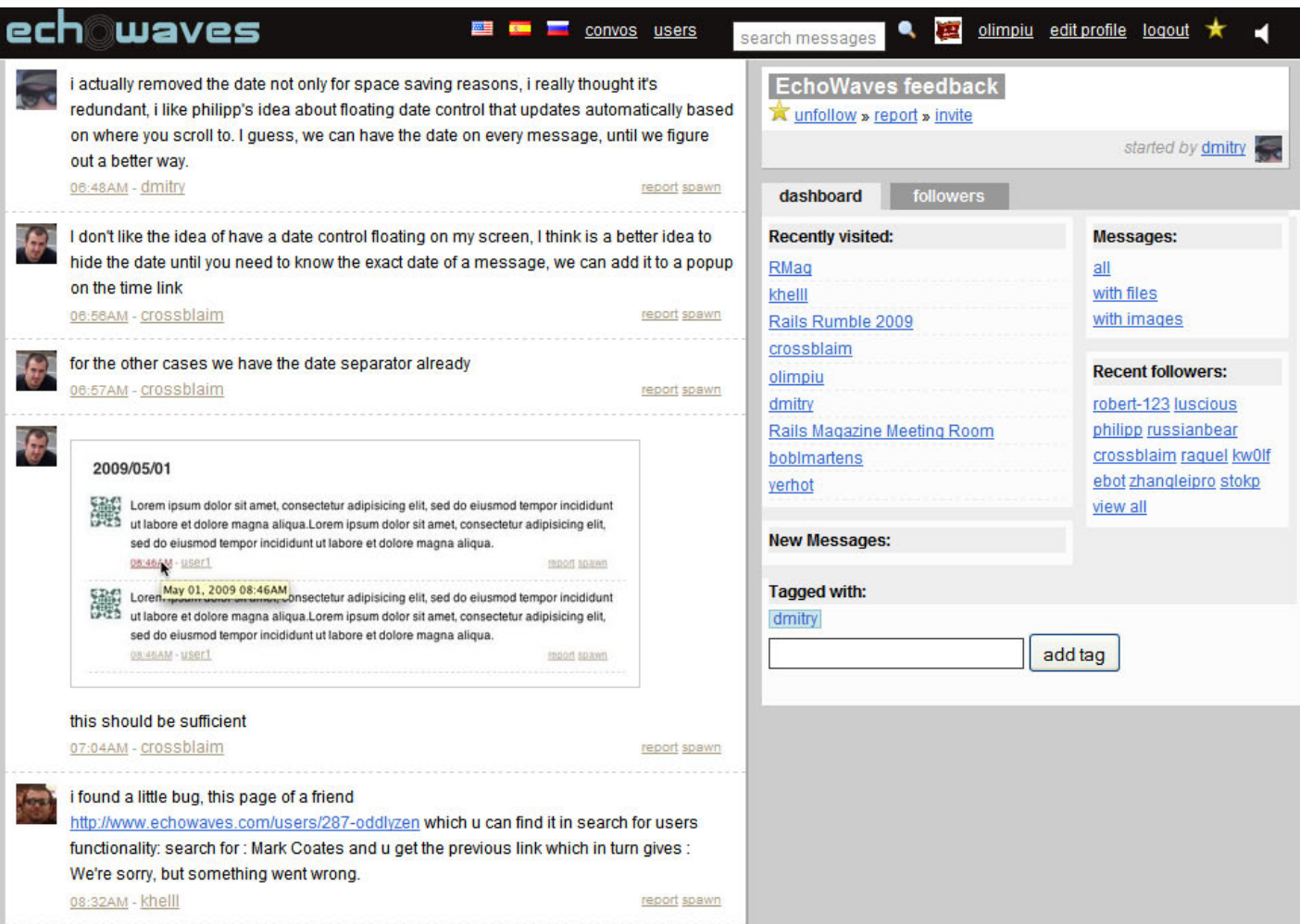
# Working on a typical Open Source Rails project

by Dmitry Amelchenko

There is a lot to say about open source (OS) development model. A typical OS Rails project has its own specifics large enough that sets it apart from the rest of other OS software projects. A typical OS project usually has a build output which can be downloaded and installed — a tar or a jar or a zip archive, a lib file, or an installable exe file, while Rails output is a Web Site. A complexity of a web application deployment perhaps is one of the reasons why OS web application development has not become a mainstream yet. Rails on the other hand opens an opportunity to develop web applications which can be easily installed on as many environments or on as many hosts as desired. Some might ask: when we are developing a web application, don't we usually want to run it on a single, well recognizable and unique host? One of the reasons for developing OS web applications would be the fact that some industries are still heavily regulated and will not allow potentially sensitive information to be sent across a wire to some web 2.0 service, but if the source code for such a service is freely available and can be installed inside corporate firewall, everyone would be happy. Another reason people choose to do OS web applications is availability of outstanding

tools and services for free in exchange for making your code open source. In this article I will be talking about some such services. And the last but not the least reason is availability of talent — if you are working on really cool OS project and utilize all the powers of the Web to promote it, the chances are that best of the best hackers will be open to work with you. What ever the reasons for people to do OS Rails projects might be, you can find a list of those in two primary sources — <http://wiki.rubyonrails.com/rails/pages/OpenSourceProjects> and <http://www.opensourcerails.com/>.

For me it all started with reading “Getting real” book by the 37Signals — it's available online at <https://gettingreal.37signals.com/>. To simply put it; the 37Signals are the best of their kind. They are by far ahead of the crowd and setting the industry standards, after all they are the Rails inventors. The book mostly talks about common sense when it comes to software development. I was also very excited to find out about some of the web 2.0 products the 37Signals offers. One of the simplest but yet useful projects that I wanted to start using right away was “Campfire” — a group chat



The screenshot shows the EchoWaves web application interface. At the top, there's a navigation bar with the "echowaves" logo, flags for US, ES, and RU, and links for "convos" and "users". A search bar labeled "search messages" is on the right, along with links for "olimpiu", "edit profile", "logout", and a star icon. The main content area displays a list of messages. The first message is from "dmitry" at 06:48AM, discussing date control. The second is from "crossblaim" at 06:56AM, discussing date control. The third is from "crossblaim" at 06:57AM, discussing date separator. The fourth is from "crossblaim" at 07:04AM, discussing date separator. The fifth is from "khelll" at 08:32AM, discussing a bug. On the right side, there's a sidebar with "EchoWaves feedback" (unfollow, report, invite), "started by dmitry", "dashboard" and "followers" tabs, "Recently visited" (RMag, khelll, Rails Rumble 2009, crossblaim, olimpiu, dmitry, Rails Magazine Meeting Room, boblmartens, verhot), "Messages" (all, with files, with images), "Recent followers" (robert-123, luscious, philipp, russianbear, crossblaim, raquel, kw0lf, eb0t, zhangleipro, stokp, view all), "New Messages:", and "Tagged with: dmitry" with an "add tag" button.



web application. But as soon I was ready to start promoting “Campfire” in the office, I realized that it’s not going to work for my company. I work for one of those firms that are very concerned about sending potentially sensitive information across the wire outside of a corporate firewall. But I was stubborn and felt that the group chat software was exactly what my company needs to improve openness and collaboration. So I decided to start hacking my own group chat web application.

At first I just wanted to write something very similar to “Campfire”. It is later, based on users’ feedback, when I realized that group chat powered by social networking aspects is even more powerful and useful. Also my intention was to get as many developers from my work place interested as possible. So opensourcing the project was not really a requirement initially. But in order to make my life easier and avoid any questions from the management, like “what do you need this server for...”, “we don’t have what you are looking for in place...”, “we can’t approve a non work related projects running on this infrastructure...”, “it has to be approved by the committee...”, “if you are doing this then it means you have a way too much time on your hand...”, I decided not to get into the corporate bureaucracy and to run the project on my own. I was going to approach the corp tech again later, when the project is off the ground and kicking.

Looking back, I think the decision to opensource the project was one of the best decisions of my life.

The next step was to figure out how to run the project and not to spend too much money out of my pocket. We live in a weird time when you can get a lot for free. And it is not like “you get what you pay for” any more — free does not mean bad at all, usually the opposite, “free” reads the best. And of course it is the best not because it is free, but because it is opensource.

As far as technology, Ruby on Rails was my first choice, free, elegant, outstanding community support — but I’m preaching to the choir here. Being a professional, I could not afford not to put the code into a version control. There are tons of options available here. Git is getting a lot of buzz lately. There are many sites that offer git repositories’ hosting. But there is something to be said about <http://github.com/>. First of all it gives 100M for public repositories for free (which really means repositories containing opensource projects). But even that is not the most important thing. What makes github stand out from the crowd is the social networking aspect. It’s crucial that your OS project is reachable. The project I started is hosted at <http://github.com/dmitryame/echowaves/> and at the moment of writing, it had 18 followers — not that many, but more then I could have possibly hoped for for a few month old project. And the cool thing is that you are not only limited to developers in your company or country. It’s

truly world wide. One of the best contributors and the technical lead on the project is located in Spain — believe it or not he found the project on github and liked it enough to join. Through github I got introduced to a guy from Britain who was also inspired by the 37Signals and started few other great projects — railscollab and rucksack, he has made a lot of very useful suggestions and had influenced the project in a big way. I can not stress enough the excitement I get from working on the project with all different people I’ve never even met in person before, located all over the world. Social networking lately has drawn a lot of scepticism, but in my experience if there is a benefit from the Internet and Social Networking, then that’s it.

**Dmitry Amelchenko** specializes in web applications development for financial companies in New York City and Boston area. Throughout his career he worked on a number of highly visible mission critical web sites as a software engineer, mentor and architect. Experienced in C++, Java and

Ruby languages. Witnessed first hand the evolution of web applications development from CGI, to J2EE, to Ruby on Rails. Currently works on number of open source projects hosted on github at <http://github.com/dmitryame/>.



As great as it sounds, it does not come for free—, there are few simple things to keep in mind. Here is what I had to do make things moving.

- Make sure to write reasonable README in you rails project. It will appear on the repo’s home page for your project on github and most often people will judge your project based on the first impression they get from that README file. So make sure it has an executive summary of your project as well as simple installation instructions. In my opinion, this is all there is to README file. If you start putting more things into it and it became long, it will distract attention and will turn people away.
- Next, make sure you have a demo installation of your project that people can play with before they make a decision to install your project on their local machines. Preferably this should be a dedicated host with a catchy name that hopefully matches the name of your project on github. My reference installation for instance is <http://echowaves.com/>. By now it’s actually more then just a reference installation. The developers that work on the application use it to collaborate on project related issues. Some people (including myself) have started photo blogs there. We get some very useful feedback from the community through it, and also using it we communicate the news and updates back

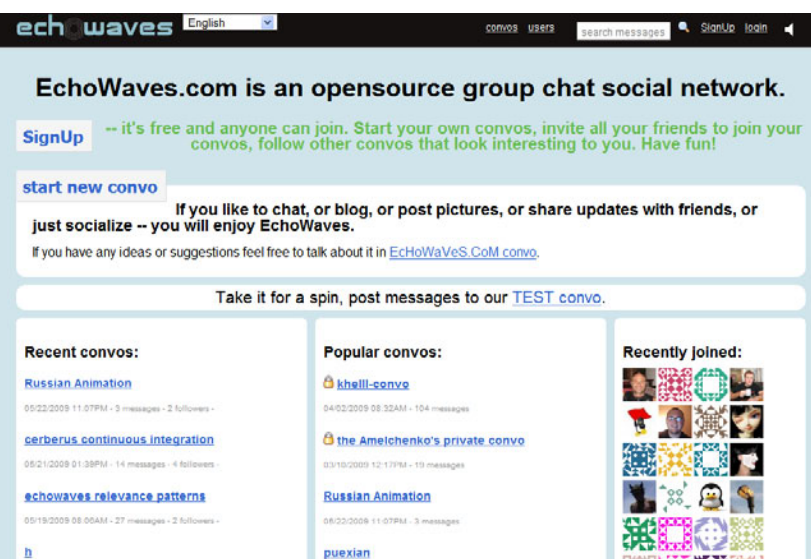
to the community. Thanks to the nature of the project, we could eat our own dog food (so feel free to checkout echowaves and run it on your own domain, or even feel free to create a conversation (“convo”) on <http://echowaves.com/>).

- Another very important thing is to make the links between your project and the github obvious. I placed a link to the github location on the default landing page and made it very explicit. And the very first link in my README file points back to the <http://echowaves.com/> site.
- Sites like github are crucial in helping to get attention to your project. The reason you are putting it on github is probably because you want to find contributors. Be open to any type of help — content writers, developers, spelling errors catchers, designers (the most difficult to find), suggestion makers. But remember: you are in charge. If you really want for your project to success, you need to have a vision. Make sure you communicate this vision whenever you get a chance, put it in on your home page, make it the first thing in your README file, talk about it in your blog. Do not start working on every little feature suggested right away. Do not accept every line of code people contribute, unless you have worked with that person before and developed a trust relationship. Read more about that in the “Getting Real” book.
- It’s essential to communicate back to the community what is currently in progress and what features are planned to be implemented. Initially I had a free account on <http://lighthouseapp.com/>, but it only supported 2 users per project (for more I had to pay and I am so cheap when it comes to OS development ;-)). Then I looked at Google Code as an alternative. Surely it supports the whole OS development stack including Subversion as a version control, but since I already started using github, I was only interested in the issues tracking aspect of Google Code. If you are more comfortable with svn, feel free to use Google Code for version con-

trol as well, but as for me I find that the social coding aspects of Google Code are just not nearly as good and helpful as github’s. The good thing is that thanks to the competition in the OS project hosting space, where you can choose what works best for you. Here is the example of how the users tracking list looks like for me <http://code.google.com/p/echowaves/issues/list/>.

The next big thing in organizing an OS Rails project is an infrastructure you will be running your site on. There are few options available here. I initially started hosting my site on a computer under my desk in a basement. But quickly realized that outsourcing the infrastructure is the better way to go. If you are serious about getting your project off the ground and want it to be noticed by other people, you have to make sure it’s up and running 24/7 and the bandwidth must be good enough to make a good impression on visitors — the web community has very high expectations these days. This is the only thing you are expected to pay for, not a whole of a lot, but still out of your pocket, so make your choice wisely. There are number of hosting providers that support rails apps like <http://www.engineyard.com/> and <http://www.slicehost.com/>. I personally choose the power and flexibility of Amazon’s web services. Read more about it here <http://aws.amazon.com/>. EC2 is easy to setup and use. They constantly add new features. They offer a FREE static IP address — a must for a serious web site hosting. They finally support persistent file system storage in addition to their S3 web service. The scaling is easy, the bandwidth is outstanding, the billing is a “pay as you go” model, so no costly contracts, no capacity planning ahead of time and no hidden costs — just use what you can afford to pay for at the moment and pay only for the things you need at the moment. In the future when the site grows above first million users and the investors start banging on your door offering loads of cash, you will be able to scale with your eyes closed. But that’s a story for another article.

After you are done setting up the project and developers start contributing their talent to the project, your reference installation is up and running and you actually have something to show, then it’s time to start promoting your project. For me the most natural place to start was at work. After all, the project was initially started as a collaboration tool for the office. I installed the app on my personal sandbox and held a lunch presentation for developers. Doing a presentation, you will be able to get an instant feedback, you will know right away if your idea is worth anything to people. During the presentation do not forget to mention that you are looking for contributors. Next, investigate your local ruby and rails community, present at one of the local Ruby or Rails groups meetings as well, if not able to present, at least send an email to the local Ruby group and introduce yourself, explain what you are doing and why, mention that you are open to suggestions and looking for volunteers. One comment about com-



munity support I have to make: if you run into a technical issues on the project, then the local community mailing list is the great resource — so do not hesitate to use it. For example, when I was looking for a library that supports chats in the browser, before spending much time on research, I asked the question on a Boston Ruby group mailing list. I've received a suggestion to use Orbited, and that ended up being one of the best architectural decision I had made on the project. Orbited is easy to install and configure and is rock solid. I actually sometimes forget that I use it at all — it works like a well oiled machine, and you never have to worry about restarting it (maybe once every few months when a new release comes out and I want to upgrade).

Depending on your project and budget, you might consider to start paid ad campaigns. I feel the majority of the OS Rails projects might not benefit from it, especially at the early stages. I tried to run Google AdWords as well as Facebook ad campaigns. The traffic has slightly increased, but the bounce rate also went through the roof. The bounce rate represents the average percentage of initial visitors to a site who “bounce” away to a different site, rather than continuing on to other pages within the same site. It's incredibly difficult to come up with the right keywords combination and even if you do find it, most of the times it does cost money. If you set your daily budget too low (like I did) your keywords will never trigger and instead Google will show your ad based on the site content, which might not be exactly what you want. Here are some more effective techniques that tend to yield better results for promoting early stage OS project (again, things are different if you have cash on your hand, then feel free to skip this part):

- Talk to all your friends, tell them about your project and ask them to post a short note about it on their blogs.
- Consider writing articles for magazines like <http://rails-magazine.com/>.
- Make sure to post on <http://www.opensourcerails.com/>.
- Post your projects status updates on Facebook, Twitter or both. This is the way to keep your friends informed about what's going on. Eventually they will notice from your status updates that something is happening, will start asking you questions, and most likely will start sharing your story with their friends.
- If you have good writing skills or really catchy idea, feel free to post on sites like <http://reddit.com/> or <http://slashdot.com/> or <http://digg.com/>, just don't be discouraged if your post does not get high ratings or does not get published at all (as in my case with).

By the time you start actively promoting your project, it helps to start monitoring your site statistics. I use Google Analytics which is free and offers a lot of different metrics.

## EchoWaves Overview

by **Khaled Al-Habache**

Echowaves is an ambitious project with many features of interest. In a way you can use it as a free Campfire-like group chatting tool. We, the RailsMagazine staff, use it to hold our regular overseas meetings using our private Convo (conversation). However that's only one side of it. Echowaves adds the social aspect to chat, making it more into Conversation based site. You can create new conversations and share your thoughts, check popular conversations and follow them and also you can follow users and check what conversations are most of interest for them. Echowaves can also be used as a Web 2.0 live forum, where topics and replies can be seen instantly. A nice thing is the ability to embed photos and videos and to attach files. This makes the conversations more interactive, also it makes it fit personal and gallery pages.

Echowaves acts as an OpenAuth provider and exports data in various formats to help developers make use of the public data.

If you like to chat, blog, post pictures, share updates with friends or just socialize — try out EchoWaves.

Running the OS project, you have to be a little bit of everything yourself; no one else will do it for you if you won't. Keep an eye on the stats and try to understand the metrics that are available to you. Simple number of hits per day is important but not nearly enough to understand what's going on. Subscribe to some free uptime monitoring service as well. I use <http://pingability.com>.

It's also as crucial to follow some basic design guidelines to get higher ratings on search engines. Things like correct page titles, meta descriptions, etc... Google webmaster tools <https://www.google.com/webmasters/tools> is of a great help in identifying some problems. If your ratings are not that high initially, do not expect it to go up very quickly, it takes a lot of patience and daily attention (it could take months). Still, make sure to analyze the suggestions Google Webmaster tool makes and try to address the issues pointed out as soon as possible.

There are probably a lot more other things you can do to make your OS Rails project a hit. The basics are still going to be the same for everyone — it is all about common sense as outlined in “Getting Real” by the 37Signals book. Your job as a project Master is to find the best possible combination that works for you. Just remember, when you are running an OS project, the most difficult thing is to make others care about it as much as you do — if you accomplish this goal, success will follow. Be passionate and honest, even if you are not the most experienced Rails developer in the world but love what you are doing, that is a sure road to success.

DISCUSS: <http://railsmagazine.com/3/3>



# Theme Support

by James Stewart

It's the ideal scenario really. You've spent many, many hours into your project, crafted the business logic so it's just right, and tuned it all to perfection. And then you realize that with just slightly tweaked views and a change of stylesheets, that same app could serve a different audience.

Or perhaps that's rather rare, but you're busy designing an app that from the outset will need to serve multiple clients, each of whom will need a bit more customization than a simple choice of logo and colour scheme provide.



**James Stewart** is a freelance web developer and consultant, based in London and primarily working with arts organizations and non-profits to develop their web presences and improve their business processes. Though officially tech-agnostic he's usually happiest building web applications with Rails or Merb. Outside of work he's enjoying the new experience of fatherhood, and dabbles in concert promotion.

In either case you could get round it with careful use of layouts, partials and other tricks from the Rails toolkit, but that can quickly become cumbersome and isn't going to scale if you need to change different elements for different clients. That's where `theme_support` plugin comes in.

Originally released by Matt McCray based on code used in the Typo blogging engine by Tobias Luetke, `theme_support` provides support for any number of themes which can override any view (whether a full view or a partial) from your main application, and have their own set of assets. So once you've got your main site set up, all you need to do is add an appropriate folder, add some declarations to your ApplicationController to identify the right theme, and override where necessary. The README file covers most of the particulars.

Before you rush out and start theming your application, there's something else you ought to know. The plugin keeps breaking. It's not that the code is buggy, but simply that goes fairly deeply into Rails, and for a variety of (very good) reasons Rails API keep changing. It's the sort of problem that we can all hope will disappear once we have Rails 3 and the defined public API, but for the time being some workarounds are necessary to get it working for subsequent Rails releases.

So in that great open source tradition, a few of us have begun experimenting with the plugin over on github. It began when I created a fork and found a workaround to serve my

fairly modest needs. Since then a couple more forks have sprung up and we're working on resolving them. I also managed to make a test suit that can help checking the state of the plugin with your installed rails version.

## Resources

Project page

[http://github.com/jystewart/theme\\_support/tree/master/](http://github.com/jystewart/theme_support/tree/master/)

Test Application

[http://github.com/jystewart/theme\\_support\\_test\\_app/](http://github.com/jystewart/theme_support_test_app/)

DISCUSS: <http://railsmagazine.com/3/4>

190 meters up





# Observer and Singleton design patterns in Ruby

by Khaled al Habache

Observer and singleton are two common design patterns that a programmer should be familiar with, however what made me write about them, is that both are there out of the box for you to use in ruby.

So let's have a look at both and see how ruby help you use them directly:

## Observer Design Pattern

According to Wikipedia:

“The observer pattern (sometimes known as publish/subscribe) is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems.”

So how does ruby help you implementing this design pattern? Well, the answer is by mixing the observable module into your subject (observed object).

Let's take an example, let's suppose we have a banking mechanism that notifies the user by several ways upon withdrawal operations that leaves the account with balance less or equal to \$0.5.

If we look deeply at this problem, we can qualify it as a good candidate for observer design pattern, where the bank account is our subject and the notification system as the observer.

Here is a code snippet for this problem and it's solution:

```
# require the observer lib file

require "observer"

class Notifier
end

class EmailNotifier < Notifier
  def update(bank_account)
    if bank_account.balance <= 10
      puts "Sending email to: '#{bank_account.owner}' informing him with his account balance: #{bank_account.balance}$."
      # send the email mechanism
    end
  end
end
```

```
end

end

class SMSNotifier < Notifier
  def update(bank_account)
    if bank_account.balance <= 0.5
      puts "Sending SMS to: '#{bank_account.owner}' informing him with his account balance: #{bank_account.balance}$."
      # send sms mechanism
    end
  end
end

class BankAccount
  # include the observable module
  include Observable
  attr_reader :owner, :balance
  def initialize(owner, amount)
    @owner, @balance = owner, amount
    # adding list of observes to the account
    add_observer EmailNotifier.new
    add_observer SMSNotifier.new
  end

  # withdraw operation
  def withdraw(amount)
    # do whatever you need
    @balance -= amount if (@balance - amount) > 0
    # now here comes our logic
    # issue that the account has changed
    changed
    # notify the observers
    notify_observers self
  end
end
```

```
account = BankAccount.new "Jim Oslen", 100
account.withdraw 99.5
```

```
#=>Sending email to: 'Jim Oslen' informing him with his ac-
count balance: 0.5$.
```

```
#=>Sending SMS to: 'Jim Oslen' informing him with his account
balance: 0.5$.
```

So to use ruby observer lib we have to implement four things:

1. Require the 'observer' lib and include it inside the subject (observed) class.
2. Declare the object to be 'changed' and then notify the observers when needed – just like we did in 'withdraw' method.
3. Declare all needed observers objects that will observe the subject.
4. Each observer must implement an 'update' method that will be called by the subject.

## Observers in Rails

You can find observers in rails when using ActiveRecord, it's a way to take out all ActiveRecord callbacks out of the model, for example a one would do this:

```
class User < ActiveRecord::Base
  after_create :send_email
```

```
private
  def send_email
    #send a welcome email
  end
end
```

a neater solution is to use Observers:

```
class UserObserver < ActiveRecord::Observer
  def after_create(user)
    #send a welcome email
  end
end
```

You can generate the previous observer using the following command:

```
ruby script/generate observer User
```

You still can have observers that map to models that don't match with the observer name using the 'observe' class method, you also can observe multiple models using the same method:

```
class NotificationObserver < ActiveRecord::Observer
```



*Hanging Low*

```
observe :user, :post

def after_create(record)
  #send thanks email
end

end
```

Finally don't forget to add the following line inside `config/environment.rb` to define observers:

```
config.active_record.observers = :user_observer
```

## Singleton Design Pattern

According to Wikipedia:

"In software engineering, the singleton pattern is a design pattern that is used to restrict instantiation of a class to one object. (This concept is also sometimes generalized to restrict the instance to a specific number of objects – for example, we can restrict the number of instances to five objects.) This is useful when exactly one object is needed to coordinate actions across the system."

The singleton design pattern is used to have one instance of some class, typically there are many places where you might want to do so, just like having one database connection, one LDAP connection, one logger instance or even one configuration object for your application.

In ruby you can use the singleton module to have the job done for you, let's take 'application configuration' as an example and check how we can use ruby to do the job:

```
# require singleton lib
require 'singleton'

class AppConfig
  # mixin the singleton module
  include Singleton

  # do the actual app configuration
  def load_config(file)
    # do your work here

    puts "Application configuration file was loaded
from file: #{file}"
  end
end

conf1 = AppConfig.instance
```

**Khaled al Habache** is a software engineer and a senior RoR engineer. A fan of open-source and big projects, and research based work. Currently giving part of his time for Ruby community and other web related work on

his blog: <http://www.khelll.com>

Khaled is a Rails Magazine contributing editor and maintains our regular Ruby column.



```
conf1.load_config "/home/khelll/conf.yml"
```

```
conf2 = AppConfig.instance
```

```
puts conf1 == conf2
```

```
# notice the following 2 lines won't work
```

```
# new method is private
```

```
AppConfig.new rescue(puts $!)
```

```
# dup won't work
```

```
conf1.dup rescue(puts $!)
```

```
#=>Application configuration file was loaded from file: /
home/khelll/conf.yml
```

```
#=>true
```

```
#=>private method 'new' called for AppConfig:Class
```

```
#=>can't dup instance of singleton AppConfig
```

So what does ruby do when you include the singleton method inside your class?

1. It makes the 'new' method private and so you can't use it.
2. It adds a class method called instance that instantiates only one instance of the class.

So to use ruby singleton module you need two things:

1. Require the lib 'singleton' then include it inside the desired class.
2. Use the 'instance' method to get the instance you need.

## Resources

Ruby observer.rb

<http://www.ruby-doc.org/stdlib/libdoc/observer/rdoc/index.html>

Singleton Module

<http://www.ruby-doc.org/stdlib/libdoc/singleton/rdoc/index.html>

Rails guides

[http://guides.rubyonrails.org/activerecord\\_validations\\_callbacks.html](http://guides.rubyonrails.org/activerecord_validations_callbacks.html)

DISCUSS: <http://railsmagazine.com/3/5>

# JRuby Tip: Monitoring with JMX

by Joshua Moore

## What is JMX?

JMX (Java Management Extensions) is a Java™ tool used to monitor and control a Java™ process. Since JRuby is a Java™ process, it can also be monitored using JMX. The great thing about JMX is that it is not just one way communication. You can also use JMX to change settings within the JRuby/Java™ process and within your own application. This article will only cover the briefest of introductions about JMX. The goal is to help you get started using it.

## JConsole

JConsole is the default JMX client provided with the JDK (Java™ Development Kit). Unfortunately, it is not part of the JRE (Java™ Runtime Environment) so if you do not have it you will need to download and install the JDK. JConsole is a simple application that connects to a Java™ process and then displays the collected information to the user. It can also be used to set variables within the Java™ process. To get started with JConsole simply execute one of these commands (depending on your OS type):

- Windows: `c:\path\to\java\bin\jconsole.exe`
- Linux: `jconsole` (it should be added to the path with the Java™ command) or `/path/to/java/bin/jconsole`
- Mac: Sorry, I cannot afford a Mac right now but I guess it would be similar to Linux (currently the Josh Moore Mac fund is accepting donations).

\* One bug with JMX on Linux is that JMX uses the IP address found in the `/etc/hosts` file to connect to the Java™ process. So if you execute `hostname -i` on the machine running the JRuby process you want to monitor, and the output is not your IP address, then you will need to modify the `/etc/hosts` file and then restart your network process or restart your computer.

\* You should be aware that the information gathered by JConsole is session based, so when the connection is closed all the gathered information will be lost.

## Setup the Jruby/Java™ process

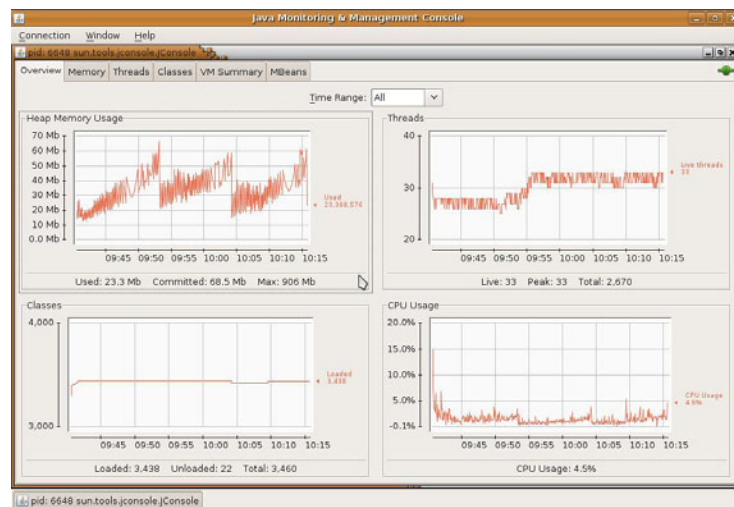
Now that JConsole is running, you will need to connect to a process so that you can monitor it. With JMX you can connect to any local process without any change to it\*. Simply click on the process and click connect (see image). Once connected you will be shown information that is being collected in real time. The overview tab gives several top level graphs of the current system resource usage (Memory, CPU, etc). The Memory, Threads, Classes, and VM Summary tabs all provide more information about each respective topic. The MBeans tab is special as it shows you the raw output received from



the JMX connection. In addition, if you want to change any of the Java settings, use this tab. It depends on the version of JConsole that comes with JDK.

\* If you are using JConsole from versions earlier than 6 then you will need to set this variable in Java:

`-Dcom.sun.management.jmxremote (java -Dcom.sun.management.jmxremote)`



or for JRuby use:

`jruby -J-Dcom.sun.management.jmxremote`

Monitoring local processes is not all that exciting. Much more interesting and useful is monitoring processes on re-



mote servers. Because of security reasons if you want to use JMX to monitor a remote JRuby process you will need to pass in some variables to the Java™ process when it starts. To be simplistic, simply start the Java™ process like this (you can choose any port number you want):

```
java -Dcom.sun.management.jmxremote.port=9999 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false
```

These options are used with Java™ or a Java™ application server like Glassfish, Tomcat, Jetty, etc. In order to monitor a process started with JRuby you will need to prepend the variables with -J :

```
-J-Dcom.sun.management.jmxremote.port=9999 -J-Dcom.sun.management.jmxremote.authenticate=false -J-Dcom.sun.management.jmxremote.ssl=false
```

Once the process is started with these variables you are ready to go.

\* You should be aware that when starting a Java™ process with these options you will be opening a huge security hole in your server. Unfortunately, I do not have time to cover the secure setup. Please see the resource section for an article on this topic.

Once the process is up and running in the JConsole connection box, click the “Remote Process” radio box and then enter in the host name or IP address followed by the port number (i.e. localhost:8004). Once the connection is established you can use JConsole in the same manner that you would for a local process.

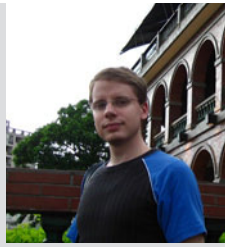
### Joshua Moore

grew up on a farm in Pennsylvania, USA. On the farm he learned that hard work is not easy and that he did not like to get dirty. So he started down the path of computer programming (less dirt, but still hard work). He liked Java and worked mostly with desktop application development until about a year ago, when he decided he could not ignore web development any longer (he ignored it before because he did not want to fight different browsers to get his applications to work right). But, different browser rendering or not, here he is using Ruby on Rails, mostly JRuby on Rails, and loving it. He chose Rails because it was simple, powerful, and no configuration needed.

Check out his blog: <http://www.codingforrent.com/>,

Twitter: @kaiping, email: [josh@codingforrent.com](mailto:josh@codingforrent.com).

Josh is a columnist with Rails Magazine, where he publishes regularly on JRuby.



## Conclusion

This is a quick introduction to JMX. This article has barely scratched the surface of what JMX can really do. The real power of JMX can be leveraged in many different ways. Some of these ways include: writing your own MBeans to collect or set custom information and writing your own JMX client to log the performance of you application. For further information on these advanced topics, checkout the resource section

## Resources

MX home page

<http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/>

More Information and JMX Authentication

<http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html>

jmx4r a gem to create a JMX client in JRuby

<http://github.com/jmesnil/jmx4r/tree/master>

jmx a gem to create a JMX client and custom MBeans

<http://ruby.dzone.com/news/jmx-gem-released>

How to write custom MBeans

<http://docs.sun.com/app/docs/doc/816-4178/6madjde4b?a=view>

Using JConsole (also explains what all the different Java™ memories mean)

<http://java.sun.com/javase/6/docs/technotes/guides/management/jconsole.html>

DISCUSS: <http://railsmagazine.com/3/6>

# Workflow solutions using AASM

by Chee Yeo

## Workflow Definition

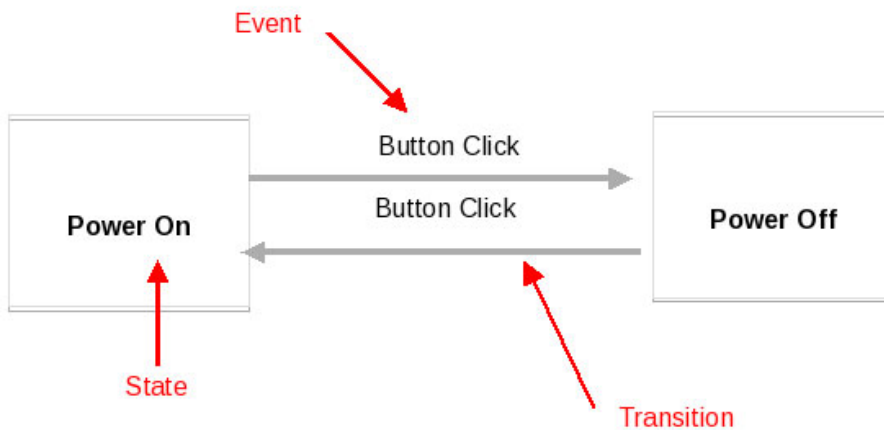
A workflow is an abstract representation of real work undertaken by a single or complex group of individuals or mechanisms. It describes a system of activities enabled through a collection of resources and information flows. Workflows are primarily used to achieve some form of processing intentions, such as data processing.

There are two main forms of workflows: sequential and state-machine. Sequential workflows are predictable. They utilize the rules and conditions we provide at the beginning to progress a process forward. The workflow is in control of the process.

A state-machine workflow is the reverse. It is driven by external events to its completion. We define the states and required transitions between those states. The workflow sits and waits for an external event to occur before transitioning to one of the defined states. The decision making process happens externally outside of the workflow – there is a structure to be followed still like a sequential workflow but control is passed to its external environment.

## What is a state machine?

The diagram below is a simple example of a state machine. It has three main properties: states, events and transitions.



## Event

A state represents a particular situation. The state machine above has two main states – ‘Power On’ and ‘Power Off’. The machine will be in one of these two states at any point in time.

There is only one main event in the state machine above – a button click. An event describes an external stimulus or input into the system. In this instance, a button click will either ‘power on’ the state machine or power it off and both states shown above respond to the same event.

A transition moves the state machine from one state to the next. In the diagram above, following a button click, if the machine is in a ‘power on’ state it will transition to the ‘power off’ state and vice versa. Not all transitions move the state machine to a new state – it is possible to loop back to the same state.

A state machine does not just store states and events alone. It can also execute code when an event arrives. In the example above, the state machine can be designed to control electricity flow when it transitions to a new state following a button click event.

## State Machine in Rails

Sometimes you need to track the state of a model in an application and the transitions between a set of pre-defined states which are triggered based on certain events. Or you might need to express the business logic of your application using a workflow to illustrate the flow of state changes based on its events.

For example, you might have an application that allows the user to upload files for processing and you need to provide some form of mechanism to track the file while it is added to a queue for processing and to provide some form of feedback to inform the user of the status of the outcome (success or an error) and the relevant action to take based on the outcome (if an error occurred, inform the user).

This can be modelled by adding a ‘state’ or ‘status’ column in your model table and updating it as you go along. However, it becomes tedious and error prone when multiple states are involved.

The `acts_as_state_machine` (AASM) plugin converts any model into a Finite State Machine capable of remembering which state it is in at any point in time and the transitions it is allowed to go through between those states and the actions that are to be triggered when a specific state is reached.

## Setting It Up

To utilise the plugin in a Rails application, simply install it:

```
ruby script/plugin install http://elitists.textdriven.com/
svn/plugins/acts_as_state_machine/trunk/ acts_as_state_machine
```

You would also require a state string column in the model table – this is where AASM stores the state of the object.

## An Example

For illustration purposes, say we have an application which allows the user to upload video files for processing and there is a Video model.

Within the video.rb model add the following code to instantiate the plugin:

```
# inside Video.rb
```

```
Class Video < AR::Base
```

```
  acts_as_state_machine, :initial => :original
```

```
  state :original
```

```
  state :processing
```

```
  state :processed, :enter => :set_processed_attributes
```

```
  state :error, :enter => :inform_user
```

```
  event :convert do
```

```
    transitions :from => :original, :to => :processing
```

```
  end
```

```
  event :converted do
```

```
    transitions :from => :processing, :to => :processed
```

```
  end
```

```
  event :failure do
```

```
    transitions :from => :processing, :to => :error
```

```
  end
```

```
  def set_processed_attributes
```

```
    # update name of file; move file to permanent storage
    location etc
```

```
  end
```

```
  def inform_user
```

```
    # pass message to user to inform of error with error code
```

```
  etc
```

```
  end
```

```
end # end of Video model
```

The `acts_as_state_machine` directive includes the plugin with an initial state of `:original`. When a new video file is uploaded





and a new video object created, this is the first state it will be in. Note that although the plugin looks for a column called state in the database table, this can be overridden in the directive like so:

```
acts_as_state_machine, :initial => :original, :column =>
'my_state_column'
```

The diagram below illustrates the current state machine of the video model.

The three main components which the AASM plugin needs to make it work are states, events and callbacks.



**Chee Yeo** is a 30 year old open source enthusiast from the United Kingdom. He is currently a Ruby on Rails developer and has produced and launched small to medium sized applications. He enjoys blogging and writing articles about Rails and contributing to open source.

Chee is also the founder of 29 Steps, a web agency specializing in Ruby, Rails and web development. See how they can help at <http://29steps.co.uk>.

## States

The possible states an object can be in are defined using the state keyword. The name of the state is expressed as a symbol. This generates an additional method of <state>? to query the state of the object:

```
@video.processed? # returns true or false
```

## Events

Events move or transition the object from one state to the next. These are declared in the example above using the event action:

```
event :convert do
  transitions :from => :original, :to => : processing
end
```

This automatically adds an instance method of <event>! to the Video class as an instance method. To call an event, simply call your object instance with the event name followed by an exclamation mark.

In our example above, suppose we have a method which picks a video out of a queue for processing:

```
def some_method
  begin
    @video.convert! # calls convert event; moves video from
'original' state to 'processing' state

    # video has been successfully converted

    @video.converted! # calls converted event; moves video
from 'processing' state to 'processed' state

  rescue

    @video.failure! # calls the failure event

  end
end
```

It is possible to have multiple transitions statement in one event block – the first one which matches the current state of the object will be called.

Sometimes it might be necessary to do some validation in an event loop before moving that object to the next state. These are known as guards and can be used in an event block like so:

```
event :convert do
  transitions :from => :original, :to => : processing,
    :guard => Proc.new { |video| video.accepted_format? }
end
```

The example above ensures that the uploaded file is of a specific format via the accepted\_format method and state transition will not be continued if it fails (if it returns false).

## Callbacks

The :enter action at the end of a state statement is used to define a callback. Callbacks are defined to trigger code once the object is transitioning into that particular state. These could be either methods or Proc objects. If a callback is defined as a symbol, like in the example above, the instance method of the object with the same name will be called.

In the Video example, when the object has reached the 'processed' state, it fires a method called set\_processed\_attributes which updates the object accordingly. There are two other actions which could be used in a state declaration: :after and :exit. The after callback gets triggered after the model has already transitioned into that state and exit gets called once the object leaves that state. We could make our Video example more expressive by defining two additional callback methods to be triggered once the object has reached the processed state:

```
state :processed,  
  :enter => :set_processed_attributes,  
  :after => Proc.new {|video| video.move_file}  
  :exit => :delete_tmp_files
```

In the example above, I declared a Proc which calls an instance method of `move_file` after the object has entered the processed state and its attributes have been updated. The instance method `delete_tmp_files` removes all temporary video files after processing and will be called once the object leaves the block. One can create very complex states and behaviours by delegating callbacks to relevant actions within a single state.

## Conclusion

The AASM plugin is a great tool to have when you need to implement state based behaviours in your application. Although the same could be achieved by rolling your own, the expressiveness of the plugin syntax itself makes it really straightforward to understand.

DISCUSS: <http://railsmagazine.com/3/7>

## Resources

[http://elitists.textdriven.com/svn/plugins/acts\\_as\\_state\\_machine/trunk/](http://elitists.textdriven.com/svn/plugins/acts_as_state_machine/trunk/)  
[rails.aizatto.com/2007/05/24/ruby-on-rails-finite-state-machine-plugin-acts\\_as\\_state\\_machine/](http://rails.aizatto.com/2007/05/24/ruby-on-rails-finite-state-machine-plugin-acts_as_state_machine/)  
[justbarebones.blogspot.com/2007/11/actsasstatemachine-enhancements.html](http://justbarebones.blogspot.com/2007/11/actsasstatemachine-enhancements.html)  
<http://github.com/ryan-allen/workflow/tree/master>

# 29 Steps. *"We love the web ... and Rails."*

We are a startup company who specializes in  
Ruby, Rails and the web.

Visit us at <http://29steps.co.uk>

Ruby . Ruby-on-Rails. Iphone. Web Design. Application Development.  
Audit. User Interface Design. Ruby-On-Rails training.

# Ruby Web Frameworks: A Dive into Waves

by Carlo Pecchia

In this article we introduce the Waves framework, mainly based on the concept of “resources” to provide web applications. No prior knowledge is needed in order to understand everything, though some experience with Ruby and Ruby On Rails could help.

## Introduction



Being a ruby programmer implies, more or less, to being also curios, to love experimentations... so we like to explore emerging web framework, particularly the ones based on Ruby.

Now it's time for *Waves*. It's defined by its developers as a Resource Oriented Architecture (ROA) framework.

But what does that mean?

Starting from the following facts, they derive that the MVC way is *only one* of the possibilities to build and code web applications:

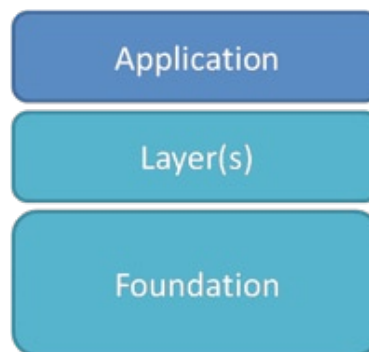
- The Web itself is based on resources, and the spreading of REST over other “protocols” is a clear symptom of that.
- Reasoning in terms of resources give you the possibility to use all the (huge) existing infrastructures of the web (caching, URI, MIME types, RSS, and so on).
- When properly structured, a ROA application can be less costly to maintain and to integrate with other applications.
- After all, HTTP has demonstrated to be a good choice for that: a protocol that calls *methods* on *resources*.
- Methods: GET, PUT, POST, DELETE, ...
- Resource: “things” identified and accessed by an URI

So, the most interesting things about this framework can be summarized here:

- Embrace of **resources** approach: the key of Waves are resource, and the routing features heavily work with that.
- The application behaves like a **proxy of resources**: we can have multiple request paths invoking the same set of resources.
- It's based, like Rails (at least with 2.3 version) and Merb, on *Rack*.

- It runs on JRuby.

In Waves we can see an application basically as a composed stack of functionalities:



- Application: is built on Layers
- Layers: mix-in features to help build your application (eg: MVC support)
- Foundation: is a Layer that provides at least the minimal features set required for an application.

In fact, Waves authors speak in terms of **Layers** and **Foundation**. Of course there aren't any magical approaches on that, we can do almost the same with Rails (or *Merb*, *Sinatra*, *Ramaze*, etc), but this “change of mind” is interesting and worth experimenting a little.

In this article we'll explore the surface of the framework that is still on version 0.8.x. After installing it, we'll write a simple application in order to illustrate how Waves works.

## Installation

Of course a gem is provided, so we simply need to type:

```
$ sudo gem install waves
```

and adding some dependencies if they aren't already present in our system.

## A simple application

Let's create a simple application for managing personal ToDo's list.

```
$ waves generate --name=todo_manager
--orm=sequel
```

Waves can use whatever ORM we prefer, in this case we choose to experiment with *Sequel*.

Previous command generates our application skeleton:

```
Rakefile
startup.rb
configurations/
```



```

controllers/
helpers/
lib/
  tasks/
models/
public/
resources/
schema/
  migrations/
templates/
tmp/
views/

```

As we can see the directory structure looks familiar to Rails one. Particularly important is the `startup.rb` file, when a Waves application starts:

```

## startup.rb
require 'foundations/classic'
require 'layers/orm/providers/sequel'

```

```

module TodoManager
  include Waves::Foundations::Classic
  include Waves::Layers::ORM::Sequel
end

```

Now, let's create a migration for our basic resource:  
\$ rake schema:migration name=create\_items

that generates the file `schema/migrations/001_create_items.rb` we'll fill with:

```

## schema/migrations/001_create_items.rb
class CreateItems < Sequel::Migration
  def up
    create_table :items do
      primary_key :id
      string :name
      string :content
      string :status
      timestamp :updated_on
    end
  end

  def down
    drop_table :items
  end
end

```

As told before here we are using Sequel ORM, but of course we can choose whatever fits our needs (ActiveRecord, DataMapper, etc).

Now we can launch the migration with:

```
$ rake schema:migrate
```

But wait! Where is the database configuration?! All configuration options live in `configurations/*.rb`, namely:

```

default.rb
development.rb
production.rb

```

**Carlo Pecchia** is an IT engineer living in Italy.

Its main interests are related to Open Source ecosystems, Ruby, web application development, code quality. He can be reached through *his blog* or on *Twitter*.

<http://carlopecchia.eu>

<http://twitter.com/carlopecchia>



Development and production modes inherit from default setups. In development we find:

```
## configurations/development.rb
```

```
module TodoManager
```

```
  module Configurations
```

```
    class Development < Default
```

```
      database :adapter => 'sqlite', :data-
      base => 'todomanager'

```

```
      reloadable [ TodoManager ]
```

```
      log :level => :debug
```

```
      host '127.0.0.1'
```

```
      port 3000
```

```
      dependencies []

```

```
      application do
```

```
        use ::Rack::ShowExceptions
```

```
        use ::Rack::Static,
```

```
          :urls => ['/
          css/', '/javascript/', '/favicon.ico'],

```

```
          :root =>

```

```
        'public'

```

```
        run ::Waves::Dispatchers::Default.new

```

```
      end

```

```
      server Waves::Servers::Mongrel

```

```
    end

```

```
  end

```

```
end

```

And in fact, after running the migration a new file containing the Sqlite development database is generated (todoman-ager).

Now it's time to put in some HTML and CSS code.

We do it using *Markaby*, the default system assumed by Waves (but of course we can switch to another templating system). It's an interesting way of generate HTML code by only writing Ruby snippets.

```
## templates/layouts/default.mab
doctype :html4_strict
html do
  head do
    title @title
    link :href => '/css/site.css', :rel =>
'stylesheet', :type => 'text/css'
  end
  body do
    h1 { a 'ToDo manager', :href => '/items'
}

    div.main do
      layout_content
    end
  end
end

## public/css/style.css
/* it's up to you to put some content here...
*/
```

Ok, it's finally time to start Waves server to ensuring us everything is set up properly:

```
$ waves server
```

And pointing out browser on <http://localhost:3000> we see the 404 error:



Of course, we haven't yet defined any request path pro-

cessing. Let's do it in main entry point, that's the resources/map.rb file:

```
## resources/map.rb
module TodoManager
  module Resources
    class Map
      include Waves::Resources::Mixin

      on( true ) { "Hi, I manage your To-
Dos..." }
    end
  end
end
```

Basically here we are saying that all request are served by a string as content. Let's see if, and how, it works in practice:

Let's add one line:

```
on( true ) { "Hi, I manage your Todos..." }
on( :get ) { "Hi, I manage your Todos...
again!" }
```

Refreshing the browser we can see the second string rendered. This is how, basically, routes work in Waves. At the end of this article we'll show how we can use this nice DSL (domain specific language) here to express routes.

## Introducing RESTful behaviour

Ok, now we have to define routes for "items" like that:

- GET /items displays the items list (of things to do...)
- GET /items/:id displays the item identified by :id
- POST /items creates a fresh new item
- PUT /items/:id modifies item identified by :id
- DELETE /items/:id deletes item identified by :id

But first we need to define Item model:

```
## models/item.rb
module TodoManager
  module Models
    class Item < Default
```

```
      before_save do
        set_with_params(:updated_on => Time.
now)
      end

      after_save do
        set_with_params(:name => self.id)
      end

      def date
        updated_on.strftime('%d.%m.%Y')
```

```

end

end

end
end

```

Due to implementation of `find` methods for the controller (maybe broken?), we have to define a `name` column in the model, and be sure it acts like an `id` field.

Let's insert some data using the console:

```

$ waves console
>> t = TodoManager::Models::Item.new
=> #<TodoManager::Models::Item @values={}>
>> t.content = 'finish waves introductory
article'
=> "finish waves introductory article"
>> t.status = 'todo'
=> "todo"
>> t.save
=> #<TodoManager::Models::Item @
values={:content=>"finish waves introductory
article",
      :status=>"todo", :id=>1, :updated_
on=>Mon Dec 29 12:00:00 +0100 2008}>
>> t
=> #<TodoManager::Models::Item @
values={:content=>"finish waves introductory
article",
      :status=>"todo", :id=>1, :updated_
on=>Mon Dec 29 12:00:01 +0100 2008}>
>> t.id

```

It's time to define the "correct" request mappings:

```

## resources/map.rb
module TodoManager
  module Resources
    class Map
      include Waves::Resources::Mixin

      # redirect '/' request do '/items'
      on ( :get, [] ) { redirect '/items' }

      # 'GET /items'
      on ( :get, ['items'] ) do
        items = controller(:item).all
        view(:item).list(:items => items)
      end
    end
  end
end
end
end

```

It's clear that the first path declaration redirects to `/items`, that is handled by the second path, when we invoke the render of `templates/item/list.mab`:

```

## templates/item/list.mab
layout :default do
  ul do
    @items.each do |i|
      li do
        i.content + ' (' + a('view',
:href => "/items/#{i.id}") + ')
      end
    end
  end
end
end

```

### A little explanation.

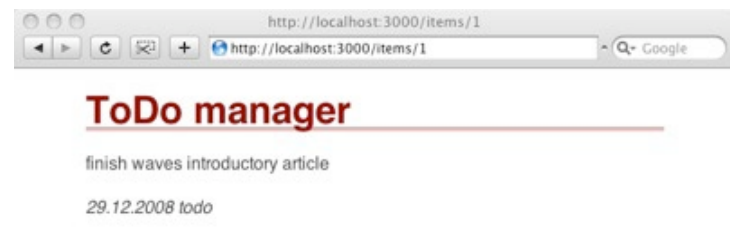
When defining the route, we see that controllers are very tiny pieces in Waves: they simply tie the request parameters to the appropriate ORM model. Don't bother because we see "controller logic" placed in `resources/map.rb`, after we'll put it in its own space.

The line:

```
view(:item).list(:items => items)
```

tells the controller to render the file `list` (with appropriate extension) found under `templates/item` (coming from `:item`) passing it a parameter named `:items` populated with some values (retrieved by the controller)

Pointing our browser on `"http://localhost:3000/items"` we can see:



### Keeping separate concerns: resource delegation

The only resource in our application is `Map` and that handles requests for "items" too. Of course we should avoid that and delegate some route paths to appropriate resource.



So, let's generate a resource for items:

```
## resources/item.rb
```

```
module TodoManager
```

```
  module Resources
```

```
    class Item < Default
```

```
      # 'GET /items'
```

```
      on(:get, ['items']) do
```

```
        items = controller.all
```

```
        view.list(:items => items)
```

```
      end
```

```
    end
```

```
  end
```

```
end
```

and keep resources/map.rb smaller:

```
## resources/map.rb
```

```
module TodoManager
```

```
  module Resources
```

```
    class Map
```

```
      include Waves::Resources::Mixin
```

```
      # redirect '/' request do '/items'
```

```
      on ( :get, [] ) { redirect '/items' }
```

```
      # delegate to item resource any path
      like '/items'
```

```
      on (true, ["items"]) { to :item }
```

```
      # delegate to item resource any path
      like '/items/<something_here>'
```

```
      on (true, ["items", true]) { to :item }
```

```
    end
```

```
  end
```

```
end
```

Hitting the refresh button on our browser shows us that everything works as expected.

Now we can inject the full REST behaviour to our "item" resource.

Let's add the handling for GET /items/<id> in resources/item.rb:

```
# 'GET /items/<id>'
```

```
on(:get, ['items', :name]) do
```

```
  item = controller.find(captured.name)
```

```
  view.show(:item => item)
```

```
end
```

And see how it is rendered in our browser:



## ToDo manager

• finish waves introductory article ( [view](#) )

So it's easy to continue defining all the other "REST methods" (exercise for the readers!). Don't forget that the PUT and DELETE methods are not currently supported by the browsers, so one of the best way is to mimic them with POST and some hidden parameters (that is the way it is done in Rails, for instance).

Basically here we can see how routing works. A path is defined as an array of "chunks":

```
# /items/past/1234
```

```
[ "items", "past", { :name => /\d+/ } ]
```

```
# /items/some-slug-here/and_something_else
```

```
[ "items", "past", :name ]
```

## Conclusion

As promised before, we'd like to shortly show how the DSL for defining routes work. Well, it's based on the concept of "Functor":

```
fib = Functor.new do
```

```
  given( 0 ) { 1 }
```

```
  given( 1 ) { 1 }
```

```
  given( Integer ) do |n|
```

```
    self.call( n-1 ) + self.call( n-2 )
```

```
  end
```

```
end
```

So we can easily create a "request functor", that is a functor defined on a resource class to match a given request pattern.

So, for example:

```
class Item
```

```
  include Waves::Resources::Mixin
```

```
  on( :get, ['name'] ) { "default item here" }
```

```
end
```

With that it is easy to define routes based on parameters matching (an expressive technique used, for example, in languages like Erlang).

Waves is a work in progress, so we can't rely too much on it for daily activities (but maybe someone will do...). Anyways, we found it really interesting, particularly on the concepts of layers: we can use MVC pattern if needed, but we are not forced to mold our solution on it.

Moreover, a strong "separation of concerns" can be done at resource levels clearly assuring, for example, that some filters are applied for a given set of resources.

Happy coding!

DISCUSS: <http://railsmagazine.com/3/8>

### Resources

Dan Yoder's talk at RubyConf2008

<http://rubyconf2008.confreaks.com/waves-a-resource-oriented-framework.html>

Waves

<http://rubywaves.com>

Rack

<http://rack.rubyforge.org>

Merb

<http://merbivore.com>

Sinatra

<http://sinatra.rubyforge.org>

Ramaze

<http://ramaze.net>

Sequel

<http://sequel.rubyforge.org>

Markaby

<http://github.com/why/markaby>

### Setting a Temporal Mark



# How to Implement Automated Testing

by Eric Anderson

You will find no shortage of discussion on which tools you should use to implement your automated test suite. What's often lacking is discussion of exactly how and what to test. The goal of this article is to provide this information.

## Determine Your Goals

Testing isn't a cookie cutter process. Each project has its own unique goals and those goals will determine what is important to test and what is not.

Moreover, testing is neither good nor bad. It's only a tool to help you reach specific goals. Some people treat automated testing like a moral decision. They insist that if you don't have 100% code coverage or practice TDD then you are a bad developer. This is an unhelpful view.

Strategies like 100% code coverage may be useful in meeting your goals. But if they aren't helping you meet your goals, don't waste valuable time implementing them just because someone else makes you feel bad for not doing so.

So let us get specific on goals. What considerations should affect your testing goals?



**Eric Anderson** does independent contract work out of Atlanta, GA. He has been developing Rails applications professionally for several years. You can find out more about his work at <http://pixelwareinc.com>.

## Desired Reliability Level

Are you writing software for a bank website, a community website, a personal website, or an internal utility? Depending on your project you will have different reliability needs.

Let us not kid ourselves. Testing isn't free. You can argue that in the long run testing will save time and money (and I would be inclined to agree with you) but there are a few things to consider.

- Some projects never make it to deployment. Testing can increase the cost of failure.
- Some projects have priorities other than 100% reliability.
- Occasionally, long run costs are less important as short run costs.

On the other hand, your company's entire business may depend on your application working correctly. Your com-

pany's reputation and your user's privacy may be at stake. So when you start that next project consider what level of reliability is important to the success of your project.

## Current Project Iteration

Even when developing the next flagship product for your company, you don't necessarily need 100% code coverage. It's crucial to consider the current state of development.

For example, is this a exploratory project? A minor or are you implementing version 8.0 of an established product?

In the early phases of a project you often don't know what to build. Often, you'll find that your early ideas about the project were incomplete, and some major refactoring is in order. And if you spent a lot of time making the first iteration bulletproof, you're out of luck. Not only are you going to scrap or refactor large chunks of the application, but you must also scrap or refactor all of the tests for that code.

When writing your tests, consider how confident you are that the application you are developing will be the same application after an iteration or two?

Personally, if I am doing exploratory programming, most of my tests are in my models since those are less prone to change. My controller tests are few, as application flow is quite likely to change. I do very little testing of the template output as that is most likely to change. And integration testing is non-existent at this point. On the other hand if I am developing iteration five of a well-defined project then my tests are much more complete.

## Development Process

Another consideration in determining your test goals is the environment, culture and business processes in your development shop. Are you likely to hand your code off to someone else to maintain and never see it again? Do other developers constantly pop in and out of code they are not familiar with to make changes? Or is the responsibility of a piece of code more focused around a small number of developers (maybe just you)?

If many hands are going to be touching the codebase (as in an open source project) then you may need more tests. Developers in this environment want to make their change, run some test and feel reasonably confident they didn't mess anything up.

On the other hand if someone (or a small group of people) is going to take responsibility for the code, then they will become more familiar with the application. So their tests may



be a bit lighter.

## What Do You Test?

When you get down to the actual code how do you know the best way to test? Since automated testing is so dependent on your goals, general rules are hard to define. The following represents a few common situations in which I've found it useful to implement automated testing.

### Breaking Down A Problem

Automated testing can help you break down a problem the little methods that are easily testable. For example, suppose that you want to display a list of items in a four column table. Your template code may look something like:

```
<table>
  <% for group in @objs.in_groups_of(4, false) %>
    <tr>
      <% for obj in group %>
        <td><%= obj %></td>
      <% end %>
    </tr>
  <% end %>
</table>
```

Let's say you were developing this code prior to the addition of `Array#in_groups_of` method to ActiveSupport (before Rails 1.2). So you need to develop your own implementation of `Array#in_groups_of`.

Now if you develop both this template code and the `Array#in_groups_of` method at the same time and try to test it all at once, you are really debugging two things at once. This will slow you down.

The alternative is to develop `Array#in_groups_of` first and write a series of tests for it.

Once you have the `Array#in_groups_of` working, you can begin writing the actual template code. You don't have to worry if the newly-created method is going to cause you problems while implementing the template code because it's is done and tested.

So you should write a test when it will help you break a problem down into smaller bites. It will be easier to debug than trying to debug the entire pie.

### Dealing with Edge Cases

Another good time to write test code is when you deal with edge cases such as an off-by-one situation. Even when you think you wrote the code correctly it is very easy to miscalculate an edge case. So to be confident, implement a quick automated test to see if you are catching that edge case correctly.

## Designing an API

Tests are also useful when you're designing an object interface. The best way to determine an ideal interface is to think of it from the calling code's perspective. If you were interacting with another object how would you like it to expose itself to you?

Tests are an ideal medium in which to develop these ideas. You can write code that uses your hypothetical API before the API has been developed. By thinking about it from the calling code's perspective you often are able to come up with a cleaner and more useful interface.

Furthermore, once the API is actually implemented you will already have some of the tests needed to verify its correctness!

Using automated testing as a way of designing software is what many people called test-driven design/development (TDD). It can be a very useful practice.

## What You Should Be Weary of Testing

We all know that testing is a good thing. But there are times when testing may actually detract from your project.

### 3rd Party Code

A big place I often see people over-test is in testing 3rd party code. A 3rd party library is being used and developers implement automated testing to ensure that 3rd party library works as advertised.

For example, I've seen code that will test to make sure `ActiveRecord#save` actually saves the record to the database (they use raw SQL to make this determination).

This is the wrong approach. If you lack confidence in the 3rd party code, don't add it to your application. And if you feel that you do need to test certain interactions with a 3<sup>rd</sup> party library, it's best to separate it from your own tests.

Now for a less obvious example. Consider the following code:

```
class User < ActiveRecord::Base
  has_many :projects
  belongs_to :manager, :class_name => 'User'
end
```

Do you need any automated testing for the above code? More than likely you do not. I have seen where over-enthusiastic testers will setup fixtures and ensure that the right projects are returned and the right manager is returned for a given record. But at that point are you really testing your app? Or are you testing the `has_many` DSL provided by Rails?

## Declarative Code

Another abuse of testing I often see, related to the 3rd party code issue, is testing declarative code. The previous code example was very declarative but let me give you a more subtle example.

```
class Notifier < ActiveMailer::Base
  def refer_a_friend(to, sender, url)
    recipients to
    from sender
    subject 'A referral from a friend'
    body :url => url
  end
end
```

Does the above code need automated testing? Probably not. However, if we added some logic to our method or email template, then the code becomes less declarative and more procedural. As the code does this it starts to become a good idea to add some automated tests to ensure your procedural code is behaving correctly.

## Markup Testing

Often, developers will over-test an application's output. This can make it very high maintenance to update without

providing much benefit in return.

A perfect example of this is the markup in your templates.

Testing for certain key bits of markup in your generated templates can be useful to automatically determine if your templates are being generated correctly. But it's a mistake to start testing too much markup, as it is very likely to change as the design of the application evolves.

Your automated testing will never be able to see if the page "looks" right. So, by over-testing markup, all you're doing is increasing maintenance costs when someone does want to make markup changes.

## Testing For Error Messages

Another area where developers over-test is error messages. It's a mistake to test for specific error messages. Error messages are very likely to get reworded so testing for the exact text will make your tests fragile.

Instead, try to find a way to test that an error occurred and maybe what type of error without doing character by character comparison of the error messages. Perhaps you can check a error code or verify the exception thrown was a certain type of exception.

For example, in a functional test of a controller you may want to make sure the `flash[:warning]` was set and not `flash[:notice]` when testing to ensure an error is processed correctly. The actual text of the flash message is less important. What is more important is that the correct type of flash message was set.

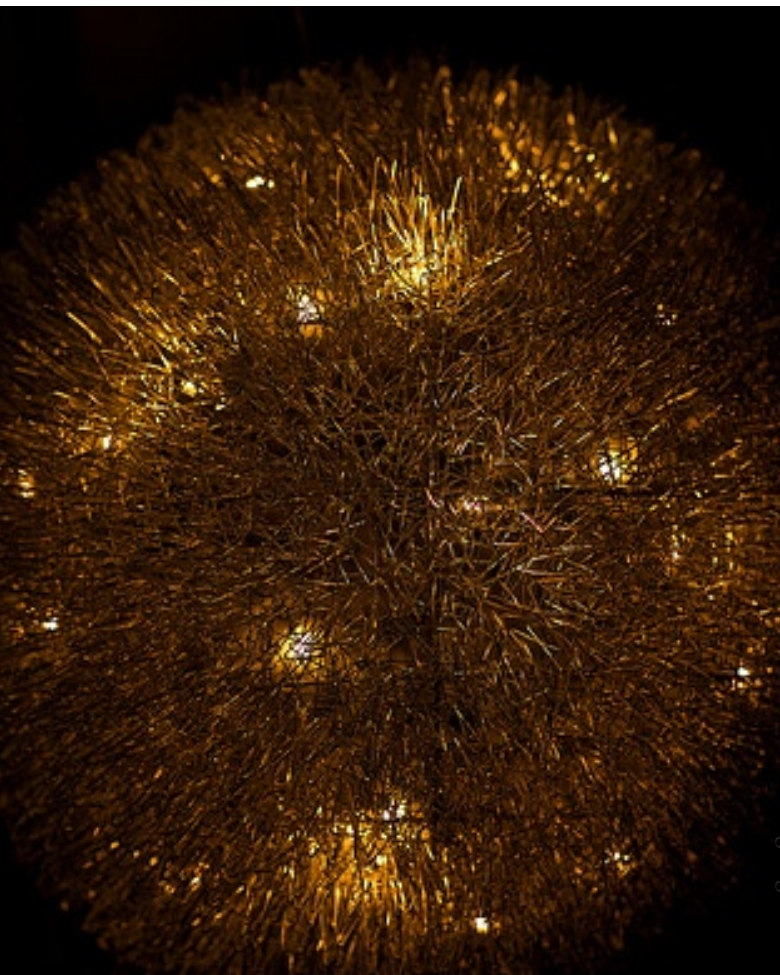
## Final Words

It's important not to be lured into thinking that because your tests pass, your application is bug-free. Absence of failure is not proof of perfection.

Automated testing should be considered a first step. Make sure to use your application in a real browser and use it often. Setup your application to report errors to you. Tools like the `exception_notifier` plugin are extremely valuable because your automated testing is only going to test for the things you can guess might go wrong. But that is really just a small fraction of what can go wrong.

In conclusion, automated testing is an excellent tool to speed up development and integration, avoid regression and gain confidence in your application's correctness. But don't fall in to the trap of blindly implementing thousands of tests just because someone says you are a bad developer if you don't. Often 100% coverage is overkill and it can sometimes even be a detriment to your application. Instead, always consider your end goals when developing a test suite.

DISCUSS: <http://railsmagazine.com/3/9>



It's going to blow

# Ruby on Rails & Flex: Building a new software generation

by Arturo Fernandez

## Introduction

A new generation of applications is gaining developers and users due to their beautiful look and feel and the great possibilities that they offer. We're talking about RIA (Rich Internet Applications), a new type of software that combines the best of web and desktop applications.

There are often times when we could benefit from combining the usability of the desktop with the features of a web application. Consider a program that organizes your MP3 collection. Obviously, it would need to access the file system. But wouldn't it also be nice if it could pull certain information from the internet, such as the artist's bio and track notes? Sounds good, don't you think?

Rich internet applications are more complex than normal web applications. Because of this, they're usually developed with the aid of a framework.

These days, we have many options to choose from, both open-source and proprietary. For Java developers, OpenLaszlo and to JavaFX are worth a look. If you prefer Microsoft technologies, there's Silverlight. But over the past two years, one framework in particular has grown very quickly: Adobe Flex.

Flex is an open source RIA framework developed by Adobe. It allows you to build RIA applications using an ActionScript-like programming language and a markup language called MXML for the interface. It was initially released in 2004 and the last version is 3 which was released in March, 2009.

Flex has two components: an SDK (Software Development Kit) and a runtime on which applications are viewed. The SDK is open source (released under the Mozilla Public License), while the runtime is the famous Adobe Flash Player which has a proprietary license.

Because the runtime is flash-based, Flex applications are cross-platform by default. And they work not only on desktops but also on many mobile devices like phones and PDAs. Moreover, the Flex SDK allows you to build software using different operating systems like GNU/Linux, MS Windows and Mac OS X since the binaries generated are cross-platform too.

While you can develop Flex applications without buying anything, Adobe does offer an additional development tool called Flex Builder. It's a modern visual IDE (Integrated Development Environment) built on top of Eclipse. Though it costs money, the Flex Builder is very useful when developing

the user interface for your application.

## Why do we need RIA applications?

A classic desktop application usually runs on a PC or Mac and it doesn't need a network connection. A good example is a word processor like Apple Pages, Microsoft Word or OpenOffice Writer. On the other hand, a web application runs on a server, and many clients simultaneously access it using web browsers.

Both types of application have their own benefits and drawbacks. Obviously, they are different from a technical point of view. Desktop applications can leverage various toolkits to create rich GUIs. They're faster. And you don't have the rendering inconsistencies that you do with browser-based applications. Web applications, however, are easy to deploy and work on any platform.

**Arturo Fernandez** is an entrepreneur, software engineer, technical writer and free/open software enthusiast. His professional experience includes services as software engineer, consultant, sysadmin and project manager working on technologies like J2EE, PHP and Rails. He currently lives in Andalucia, Spain where he founded BSNUX Software Factory (<http://www.bsnux.com>) a company specialized in RIA and mobile applications. He can be reached at [arturo@bsnux.com](mailto:arturo@bsnux.com).



Technologies like AJAX bring the web a little closer to desktop-like performance, at a cost of increased complexity. Developers are forced to work in multiple languages: JavaScript for the front end and another language like Ruby for the back end. But with Flex, you can develop an application from top to bottom in one language.

Developing a RIA application we can join the best of desktop and web applications, resulting in a cross-platform application with a good and flexible look & feel that can exchange data with other applications using standard protocols and technologies through the Internet.

Now let's take a look at how we can use Adobe Flex and Rails to develop great software in less time.

## Front-End and Back-End

A rich internet application will typically have two major components: a front-end which is the client and a back-end which contains the business logic and data.



When you write an application in Adobe Flex, you'll end up with an executable that runs on the Adobe Flash Player, either inside a web browser or on another device.

The back-end is a web application built using any technology like J2EE (Java Enterprise Edition), PHP or Rails. We'll use Rails because it's our favorite framework and because it is a good complement for Flex.

The next step is to learn how the front end communicates with the back end. Let's get to the code!

## How Flex Communicates With Rails

There are two ways to establish communication between Flex and Rails. The first is AMF (Action Message Format), a protocol to exchange data through traditional RPC (Remote Procedure Call). The second is HTTPService, a mechanism that allows Flex to make a call to the backend using the HTTP protocol.

AMF is good for mapping objects on the server with objects in the client. Using AMF we can map an ActionScript object with a Ruby object, like a model instance. It's also possible to call a controller method directly from ActionScript, sending an object as a parameter.

Let see how to use AMF and HTTPService with Flex in practice.

As an example, we'll create an application with CRUD (Create, Read, Update, Delete) operations over a database table of products. First, we'll create a front end in Flex, then a back end in Rails.

First, we need to create a new Flex's project. The easiest way to do that is to use Flex Builder. Select File > New > Flex Project. Name the project and the main MXML file rails\_magazine. Be sure to check "Web Application (Runs on Flash Player)", as shown in Figure 1. This wizard will create all of the files needed by the application.

The most important directory for us is `src/`. This is where our code files will reside. Using Flex Builder, we can build our GUI quickly via drag & drop. Widgets are on the left pane and workspace is the main area. A Datagrid is a great widget offered by Flex. It's used to display data loaded from the database and allow typical operations like sorting and filtering.

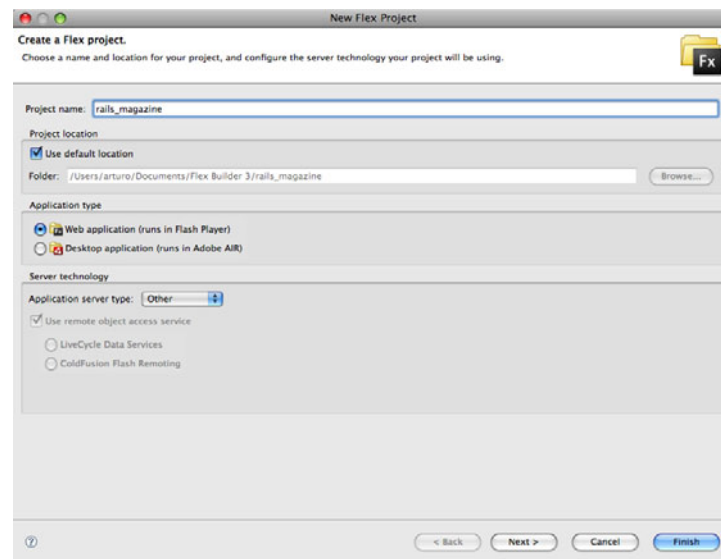


Figure 1. Creating a new Flex project

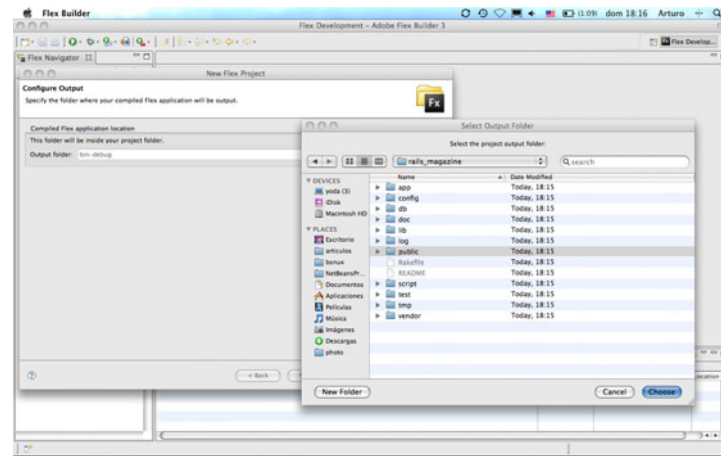


Figure 2. Selecting public directory

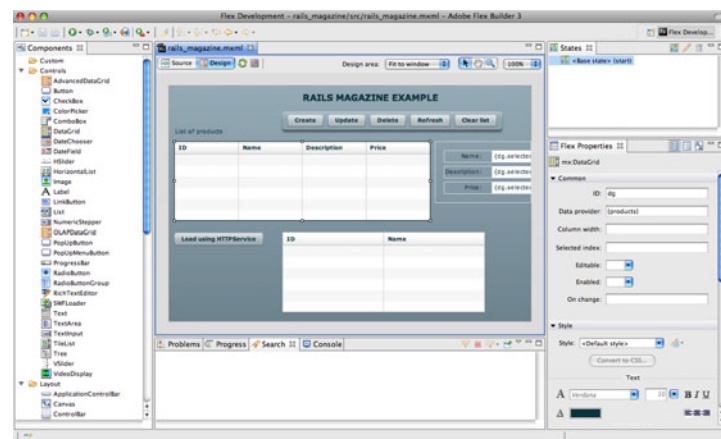


Figure 3. Building a GUI for our application

The Flex binaries have to be accessible over the web. So we'll put them in the public directory of the Rails application. This is easy, we only need to select this directory for Output



folder as shown in Figure 2. Once we're finished with the GUI, we can create the back-end.

First, we'll create a Rails application using the rails command. Second we will need a model to store the product's data so we will execute this command:

```
$ script/generate model Product name:string
description:string price:float
```

Now we need to configure our database. By default Rails 2 offers a simple configuration for an SQLite database, so we will use that. The next step is to run the migration:

```
$ rake db:migrate
```

Now we need a controller to manage the client's requests:

```
$ script/generate controller products
```

Our backend is finished for now. The next step is to configure connection between it and the front-end.

## RubyAMF

In this example, we've created two datagrids (Figure 3). One will use AMF to communicate with the server. The other will use HTTPService. We'll cover AMF First.

Fortunately for Rails developers, we have the RubyAMF rails plugin which will do a lot of the heavy lifting. Installation is easy. Just type the following:

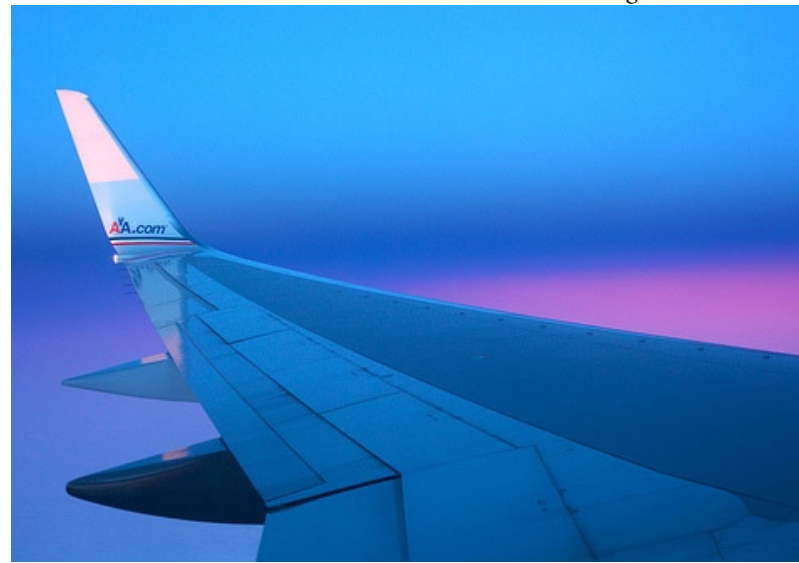
```
$ script/plugin install http://rubyamf.googlecode.com/
svn/tags/current/rubyamf
```

When installation is finished we will be ready to write a little code for our Rails application. In the config folder we can find a file called rubyamf\_config.rb that includes the mapping between our ActiveRecord class and our ActionScript class.

But we don't have an ActionScript class! Let's write one. It will map to the product model that we just created in rails. In the Flex project, create a new subfolder of src called valueobject and new ActionScript class called Product. This is the code for the new Product.as file:

```
package valueobj
{
    [RemoteClass(alias="Product")]

    public class Product
    {
        public var id:int;
        public var name:String;
        public var description:String;
        public var price:Number;
        public var createdAt:Date;
        public var updatedAt:Date;
```



```
/**
 * Constructor
 */
public function Product()
{
}
}
```

Notice that the public attributes of this class match those in our Rails model. We also declare that RemoteClass is Product – our Rails model class.

Now go back to rubyamf\_config.rb and add these lines of code:

```
ClassMappings.register(
    :actionscript => 'Product',
    :ruby => 'Product',
    :type => 'active_record',
    :attributes => ["id", "name", "description", "price",
        "created_at", "updated_at"]
)
```

Now that we've mapped the Flex Product class to the Rails Product class, we can load the data into our grid. You'll need to set the dataProvider property of the datagrid component:

```
<mx:DataGrid id="dg" dataProvider="{products}" x="10" y="100"
width="457">
    <mx:columns>
        <mx:DataGridColumn dataField="id" headerText="ID"/>
        <mx:DataGridColumn dataField="name" headerText="Name"/>
        <mx:DataGridColumn dataField="description"
headerText="Description"/>
        <mx:DataGridColumn dataField="price" headerText="Price"/>
    </mx:columns>
```

```
</mx:DataGrid>
```

products will be an array of products, so we will need an ActionScript variable to store data received from database:

```
[Bindable]
```

```
private var products:Array = new Array();
```

Note that we're using the Bindable modifier to indicate that this variable is needed by the datagrid. It's very important to indicate in Flex code that we need AMF to invoke remote methods so we will write this code:

```
<mx:RemoteObject id="productService" destination="rubymf"
  endpoint="http://localhost:3000/rubymf_gateway/"
  source="ProductsController"
  showBusyCursor="true"
  result="resultHandler(event)"
  fault="faultHandler(event)" />
```

RemoteObject is a component offered by Flex to manage communication with the backend. Properties like result and fault will invoke a different ActionScript functions. The result callback indicates what to do when the request is complete. In this case we store the result of each request in the products ActionScript array. The fault callback is for error handling. Add this code to rails\_magazine.mxml:

```
private function resultHandler(event:ResultEvent):void {
  products = event.result as Array;}

```

Remember, products is a bindable array, which is associated to our datagrid. The connection between components is automatic. It is important, however, to keep in mind that communication with the backend is asynchronous.

CRUD operations are accessible through the GUI's buttons. When clicked, each button invokes a corresponding Rails action. For example, to list all products in the datagrid we need to call the load\_all method of the controller class. To do this, we can write an ActionScript function and set it as a callback for the Refresh button:

```
private function load():void {
  var token:AsyncToken = AsyncToken(productService.load_
all());
}
```

Sending an object with data from Flex to Rails is possible too. To store data in the database this ActionScript function is needed:

```
private function save():void {
  var prod:Product = new Product();
  prod.id = dg.selectedItem.id;
```

*From Burn to Blue*

```
prod.name = Name.text;
prod.description = Description.text;
prod.price = parseFloat(Price.text);
var token:AsyncToken = AsyncToken(productService.
update(prod));
}
```

Back in our Rails controller, we need to add some CRUD methods. For example:

```
def load_all
  @products = Product.find(:all)
  respond_to do |format|
    format.amf { render :amf => Product.find(:all) }
    format.xml { render :xml => @products.to_xml }
  end
end

def update
  @product = Product.find(params[0].id)
  @product.name = params[0].name
  @product.description = params[0].description
  @product.price = params[0].price
  respond_to do |format|
    if @product.save!
      format.amf { render :amf => @product }
    else
      format.amf { render :amf => FaultObject.new("Error
updating product") }
    end
  end
end
```

We used `render :amf` to indicate that we're using the AMF protocol. Before testing our work, let's see how Flex can communicate with Rails using HTTPService.

## HTTPService

This method is simpler but less flexible than AMF. In our example we will build a new DataGrid to link it with the HTTPService component. This is the code:

```
<mx:DataGrid id="dg0" dataProvider="{productHttpService.
lastResult.products.product}"
x="203" y="262" width="361">
  <mx:columns>
    <mx:DataGridColumn dataField="id" headerText="ID"/>
    <mx:DataGridColumn dataField="name" headerText="Name"/>
  </mx:columns>
</mx:DataGrid>
```

Note that we used a different value for the `dataProvider` property. Now we need to declare the `productHttpService` component in Flex:

```
<mx:HTTPService id="productHttpService" url="http://local-
host:3000/products/load_all" />
```

HTTPService requests consume normal XML. If you take a look at the `load_all` controller method above, you'll see where the XML comes from.

Finally, we'll add a button which loads makes the HTTPService request.

```
<mx:Button label="Load using HTTPService"
click="productHttpService.send();" />
```

That's all folks!! We have all the code finished, let's check it out!

## Running

Now we are ready to test our application. Clicking in the play button of the Flex Builder generates a binary file in SWF format and a HTML page which invokes this binary directly in the public directory of our Rails application. Flex Builder opens a web browser with the URL so we can see our application, as seen in Figure 4.

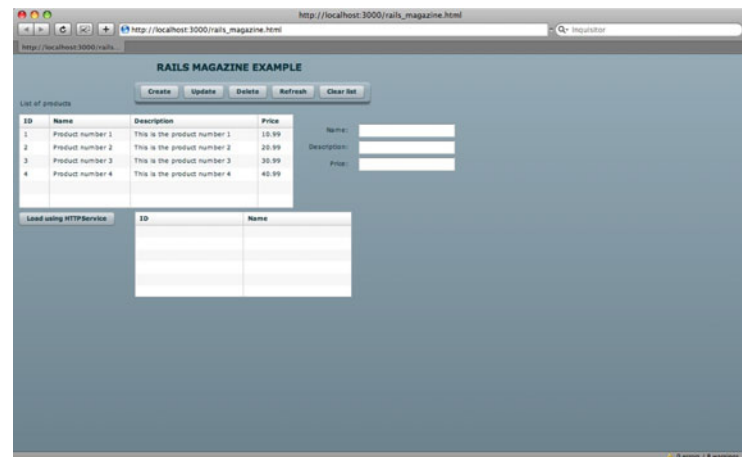


Figure 4. Running our Flex application

Feel free to click every button to see how the application transfers data from the front-end to the back-end.

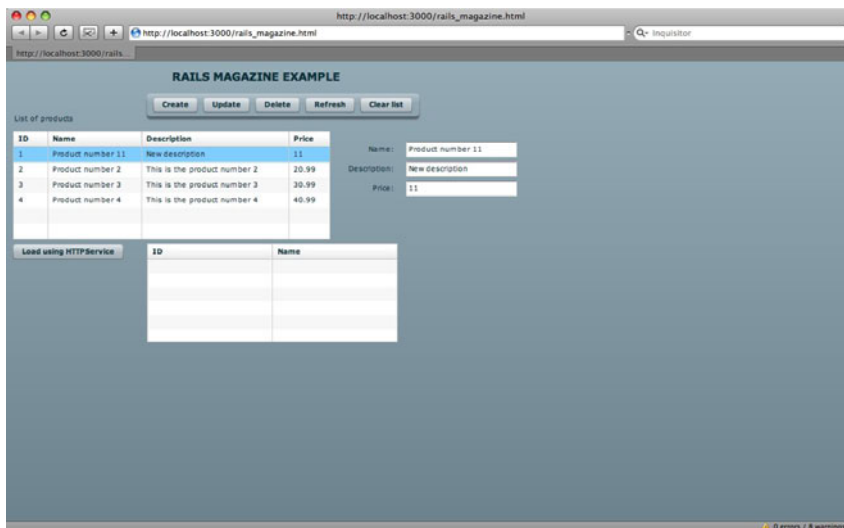


Figure 5. Showing data of a selected product

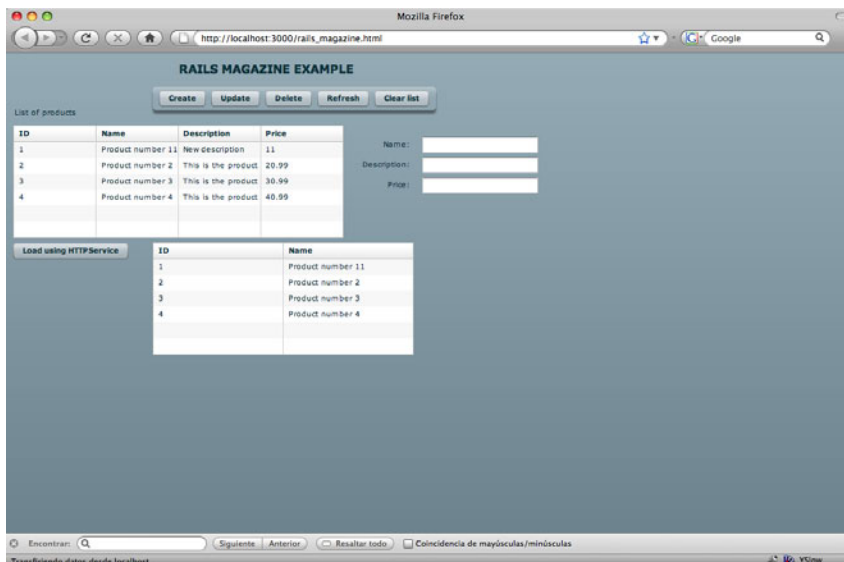


Figure 6. Loading data using HTTPService

Congratulations!! Your first RIA application with Flex and Rails is up and running.

## Conclusions

Flex is a growing technology and Rails can play a great role by providing a solid, robust and stable back-end. We've only scratched the surface of what you can do with Rails and Flex. Check out our references to learn more. Are you ready? Let's start coding!

DISCUSS: <http://railsmagazine.com/3/10>

## Resources

Flex site

<http://www.adobe.com/products/flex/>

Information about ActionScript programming language

<http://www.adobe.com/devnet/actionscript/>

Main features of Flex Builder

[http://www.adobe.com/products/flex/features/flex\\_builder/](http://www.adobe.com/products/flex/features/flex_builder/)

Complete information about MXML

<http://learn.adobe.com/wiki/display/Flex/MXML>

Blog about Flex and Rails

<http://flexonrails.net/>

Wikipedia definition of AMF

[http://en.wikipedia.org/wiki/Action\\_Message\\_Format](http://en.wikipedia.org/wiki/Action_Message_Format)

RubyAMF site

<http://code.google.com/p/rubyamf/>

OpenLaszlo site

<http://www.openlaszlo.org/>



# meshU

## Rails Magazine

### Exclusive Coverage

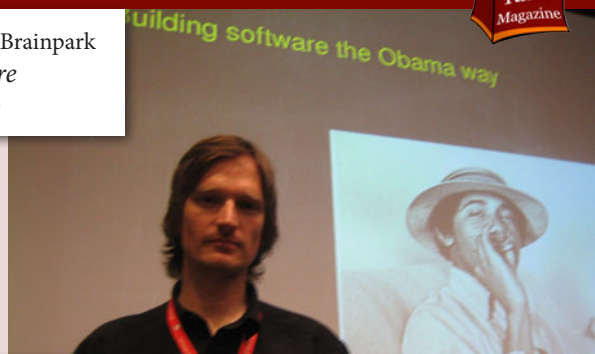
**meshU** is a one-day event of focused workshops on design, development and team management given by those who have earned their stripes in the startup game; people who can talk knowledgeably about everything from interface design to using Amazon's S3 distributed server network.

**meshU** 2009 speakers included Chris Wanstrath, Ilya Grigorik, Carl Mercier and Ryan Singer – to name just the few that we interviewed.

**meshU** was held on April 6th, 2009, at the MaRS Collaboration Centre in downtown Toronto.

The official site:  
<http://www.meshu.ca/>  
 Speakers and sessions:  
<http://www.meshu.ca/speakers-2009/>

Brydon Gilliss, Brainpark  
*Building software the Obama way*



Ilya Grigorik, AideRSS  
*Event-Driven Architectures*



Chris Wanstrath, GitHub  
*Building a Business with Open Source*



Pete Forde, Unspace  
*Is that an iPhone in your pocket, or are you just happy to see me?*





# Building a Business with Open Source

by Chris Wanstrath at meshU

Hi everyone, I'm Chris Wanstrath.

When I think of workshops, I think of 8th grade shop class. Where you fashion little letters out of wood and get to play with really big saws for 50 minutes each day. You remember, right?

My shop class was taught by Mr, let's say, Allen. Mr. Tim Allen. (It's a fake name but this is a true story.)

Mr. Allen's shop had a mini computer lab in it, which was very cool. Stocked with state of the art IBM PCs running Netscape Navigator, we could use Lycos, Yahoo, or (my personal favorite) Metacrawler to find and print out designs or wood patterns.

The computer lab was in its own room, near the back of the shop. I never understood why there was a second room, but looking back I'm pretty sure it had something to do with the combination of teenagers, electric saws, and expensive computer equipment. In science class we'd call that "potential energy."

So the computer lab, being in its own room, had a door and a window or two. The windows had blinds that you could close to prevent glare.

On days where we were independently working on an



**Chris Wanstrath** is a co-founder of GitHub (<http://github.com>) and an Isaac Asimov fan. He lives in San Francisco, never has enough time to work on his own open source projects, and tweets at <http://twitter.com/defunkt>

in-progress project, Mr. Allen would enter the computer lab, shut the door, then close the blinds. Occasionally he'd peak his beady little eyes out to make sure we were all still alive, then quickly retreat back to his work.

We always joked that he was looking at porn, but in reality he was probably looking at houses, reading the news, or (hopefully) trying to find a new job.

No. He was looking at porn.

The police got involved and he was fired the next semester.

I didn't learn much in shop class, but I did learn one thing: the Internet can be used for good or evil.

(And just to be clear: I'm not saying Mr. Allen's behavior was necessarily evil. But at school, around children? Creepy.)

8th grade, as it happens, was also when I discovered open source. I went to a Linux install-fest at the University of Cincinnati with some of my older friends and put Red Hat Linux on my family's Compaq Presario. (Sorry Mom and Dad.)

It didn't have a window system, only console mode, and was the first time in my life I felt like a badass. Like a hacker in the movies. Awesome.

Over the next few years I worked at a Linux based ISP, worked in a Windows based IT department, wrote ASP, wrote PHP, loved the GPL, hated the GPL, then eventually ended up here.

Today I work at GitHub, a company I co-founded last year. In this workshop I'm going to talk about the ways we've used the web and open source to stay cheap and help grow our business from a bootstrapped startup to a profitable company. And no porn, I promise.

Git is a version control system similar to Subversion or CVS. It was written for the Linux Kernel and is used by projects such as Ruby on Rails, Scriptaculous, YUI, Perl, Android, and the open source code that powers Reddit.

GitHub is a site that lets you publish Git repositories, either public or private. Tom Preston-Werner and I started working on it in 2007 while we both had other jobs. He was at the natural language search startup Powerset while I was running a consulting business with PJ Hyett.

Both Tom and I were big fans of open source. I had released dozens of projects and contributed to many, and Tom was the same way.

We were both so active in open source that it was beginning to take a toll. Neither of us had created any huge projects, but we had lots of little ones and the time was adding up. Managing bug trackers, reviewing and merging patches, discussions on mailing lists, writing documentation, on and on.

GitHub was our solution: a way to streamline the open source process. A way to make contributing, accepting, and publishing code stupid simple. In the same way that Django or Rails does the tedious stuff so you can focus on your application, we wanted GitHub to let you focus on your code.

At least, that's the official sales pitch version. In reality I just wanted GitHub to let me focus on my code. I've learned from experience that I'm much better at building things for

myself than I am at building things for other people.

Tom had learned from experience that if you're going to have a side project with the potential to be resource intensive, it's gotta pay for itself. His last project, Gravatar, got pretty damn popular and pretty damn expensive. He ended up spending a lot of time and money scaling it. Time he could have used for mountain biking and money he could have spent on scotch.

Unfortunately, self-sustaining websites are not easy. It's literally the million dollar question. How do we make money with this thing?

Well, how was our competition making money?

Sourceforge, a popular code hosting site, is free for open source projects but has ads. Lots of ads.

I hate ads.

I used to work at CNET on a handful of sites that were ad supported. In that model, it's all about volume. Get as many eyeballs on your site as possible and hope a fraction of them click on the ads. Deal with sales people and advertising agencies. You really need to become popular before making money.

That works for some, but not for me. And it gets worse.

The people paying your salary aren't your customers. They're the advertisers. Your business lives and dies by people who care about how much and what kind of traffic you drive, not how good your product is. They care about your demo (that's short for demographic) and cpms (I don't even know what that's short for anymore).

For example, each year Vista was delayed, CNET had to change its budget because there would be no massive Microsoft advertising campaign. Same story with Sony and the delayed Playstation 3.

The funny thing is, these companies bossing you around with their advertising campaigns aren't making money off advertising. They're selling things that people want.

It quickly became clear to me that the Microsofts and Sonys had much stronger business models than the CNETs and Sourceforages.

So free is out – this thing needs to pay for itself. And ads are out – they suck. What's left?

The revolutionary concept of charging people money.

If you want a private project on GitHub, you have to pay. If you want a public project, go nuts: it's free.

Don't ever make people pay to participate in open source. That's as bad as looking at porn in shop class.

So with the business model in place, we continued working on the site in our free time. We didn't want to put a lot of our own money into it, but we definitely wanted to keep it self-funded. It was a side project, not a startup.

At a local tech meetup we showed off the site to some friends who immediately wanted access. Shortly thereafter we launched the beta and invited them to sign up.

After that beta began we began noticing what we've dubbed "the YouTube effect." People were blogging about the cool things they were doing on GitHub – not about GitHub itself. We'd get huge traffic spikes from people writing blog posts announcing their project or idea, with a single, casual link to the project on GitHub. Even better, we'd get tiny traffic spikes in great numbers from less notable projects doing the same. More and more people were blogging about their cool project on Github. Showing off.

You didn't need a beta to use GitHub. Only to make an account and share. People with accounts could share with anyone. This was pretty key – everyone could see what was going on, they just couldn't participate without an invite.

After the YouTube-like blogging came "the Facebook effect." Once a project was hosted on GitHub it made a lot of sense to invite the other people involved. The more, the merrier.

Like the YouTube effect, this was not something we had planned for or anticipated.

Something we did anticipate, however, was our invite system. During a consulting gig I was introduced to the concept of "artificial scarcity." When you control the means of production, you can limit production to make a product seem more valuable.

Just look at the Wii. Or, more appropriately, gold in World of Warcraft.

So during the beta, our invite system was modeled after Gmail's: once you received an invite and signed up, you could then invite five others. This meant people would be asking for GitHub invites on Twitter, mailing lists, and message boards. There'd be entire threads devoted to people asking for GitHub invites. It wasn't hard to get an invite, you just had to ask.

We could have given invites to everyone who asked, but this gave us free publicity and made the invites more desirable. Hey, these invites are rare and I got one – why not give this GitHub thing a try?

Unable to afford any advertising or traditional marketing, the Gmail-style beta system worked better than we could have hoped. Combined with the YouTube and Facebook effect, we were starting to see real traction without spending any money on ads.



Looking back, there is another, more traditional term for what was happening. “Word of mouth.”

If you’re building a website, you have a huge advantage over more traditional businesses: all of your potential customers have access to the Internet. You don’t need to buy billboards, get written up in newspapers, or buy commercials on TV in the hopes of getting your name out there. The Internet provides better, faster, and cheaper means of advertising.

Best of all, it’s more trustworthy and authoritative. Friends recommend quality products to friends. It’s not some baseball player on a TV ad but a person you trust.

Somehow, this is still a secret. Companies still believe that what works offline will work online.

The longer they believe that, the better it is for all of us who know better.

However, we did get curious. We started dabbling with Google AdWords and advertising on small blogs. We sponsored regional conferences. We gave away t-shirts. Who knows – maybe we’d find a hit.

The Adword conversion rates were abysmal. I’m glad that we gave it a shot, but for our business it just doesn’t work. We’ve found people trust their peers and personal experience to find a hosting provider, not random Google ads.

Same for the blog ads. It’s nice to sponsor someone’s quality, unpaid time, but when you’re a self funded startup the dollars spent are not effective enough.

As for the regional conferences, spending money to be just another logo in a pamphlet or on a poster is not something we can afford to do. Instead we’ve started doing more guerilla style marketing: last weekend I flew to a conference in Chicago and spent the money we would have spent sponsoring it on hanging out with developers. Saturday night, for instance, I took a group out for pizza and beers.

I got to drink with GitHub users, talk about version control with people who’d never used the site, and give our website a human face. We’ve done this a few times now and are finding it to be extremely effective.

Who knew: actually meeting your customers is good for business.

The last promotion technique I mentioned was giving away t-shirts. Yeah, that’s awesome. Do that. Everyone loves t-shirts.

So we got an idea, figured out a business model, launched a beta, got users, and made t-shirts. But what about the site itself?

In our efforts to improve and expand the site, we found

open source software to be an extremely cost effective way to develop many core pieces of our infrastructure.

GitHub is built on a variant of the highly successful LAMP stack. It stands for Linux, Apache, MySQL, and PHP (or Perl (or Python)).

Our own version looks more like Linux, Nginx, MySQL, Ruby, Git, Bash, Python, C, Monit, God, Xen, HAProxy, and memcached. But none of those start with a vowel and LAMP is very pronounceable.

Basically, going with a LAMP-based stack is pretty much a no brainer unless you’re a Java or Microsoft shop, in which case you’re probably not a bootstrapped startup on a budget.

But we were, so we went with it. Running an open source stack, with a background working with open source libraries, mean you’re constantly looking for code to extract and release from your own projects.

The first thing we open sourced, very early on, was the Git interface library called Grit. There was nothing like it available at the time and it remains a very core piece of our site.

It would have been easy for us to think Grit was some magical secret sauce, a “competitive advantage,” but we now know how wrong that would have been. In fact, I’d say one of the most competitively advantageous things we did was open source Grit.

A few weeks after its release a man by the name of Scott Chacon published a competing library. His had some amazing stuff in it but lacked certain features Grit excelled at. Tom and I chatted with Scott and eventually convinced him to merge his code into Grit, creating one library to rule them all.

A few months later we hired Scott. He went on to write some of the most mind blowing GitHub features.

Good thing we open sourced Grit.

As the site grew and more businesses started using it, people began requesting integration with various third party services. They wanted IRC notifications, ticket tracker awareness, stuff like that.

Integration with established sites is a great thing to have. It’s sexy and lets people use tools they’re familiar with – we wanted to add as many as we could. Doing so, however, would be prohibitively time consuming. We’d have to sign up with the third party service, learn their API, write the integration, test it to make sure it worked, then address any bugs or complaints our customers had with it. Over and over and over again.

So we open sourced that part of the site, too.

People immediately started adding their own pet services



and fixing bugs in existing ones. We've even seen third party sites write their own service, in order to advertise "GitHub Integration" as a perk.

Everyone wins.

Needless to say, this idea became very attractive to us: open sourcing core parts of the site so motivated individuals can fix, change, or add whatever they see fit.

So we did it again with our gem builder. We host Ruby packages for development versions of code. The utility that creates the packages is now open source. We've had various bug fixes and security patches submitted, which feels good. Fixes and patches that perhaps would have otherwise been overlooked had we kept the source closed.

Open sourcing this component also means people can run the builder on their local machine before submitting a package to GitHub and have some idea of whether or not they're doing things correctly. Cool.

With the final successful project I want to talk about, we went the opposite direction. Tom created a static site generator in his free time called Jekyll. When we launched static site hosting, integrating Jekyll was obvious. Since then we've had dozens of bug fixes submitted and features contributed by people who want to use GitHub for their static site but needed just a little bit more out of Jekyll. Now they have it.

And sure, our site does cater to developers. But open source has its own community that any business can take advantage of and contribute back to.

At my last startup, we open sourced a JavaScript plugin we'd developed. I thought that people would see the plugin and get interested in the startup, maybe sign up for a trial account. Peek around. Dip their toe in.

Didn't happen. Should've known.

But! That did not stop people from using and contributing to the plugin. Even though the users of my site weren't adding features they wanted, other people were. My customers were still benefiting from open source – probably without even knowing the concept existed.

Your site doesn't need to be developer-centric to benefit from open source. Just don't count on it driving signups to your Nascar based social network. Use open source to improve the quality of your product, not as a marketing tool.

And while we're talking about things open source won't do for you, I might as well address the big question that always comes up:

Why isn't GitHub open source?

After all, WordPress is open source and still makes money

hosting blogs. Lots of money. Why not GitHub?

So, I actually love this question because the answer fits so well with everything I've been talking about. In fact, I already gave the answer.

The reason GitHub isn't open source is the same reason we started GitHub: open source takes a lot of time. And as we all know, time is money. Especially so in a small company.

Managing bug trackers, reviewing and merging patches, discussions on mailing lists, writing documentation – these are all things we'd have to do in addition to working on the site and running the business. So in addition to my current job, I'd basically need to do a second job.

And my current job already takes up a lot of my time.

With that said, it's not out of the question for the future. It would just be too expensive right now.

I believe that by worrying about every dollar spent and every dollar earned, we're a much stronger company. We didn't have an office at first because we couldn't afford one, but now that we can we still don't. We've realized that, for us, it's an unnecessary expense. We all enjoy working from home or at cafes.

Company meetings are held at restaurants over dinner and beers. At the end of the month, it's a lot cheaper than rent would be. And a lot more fun than board rooms and white boards.

Starting small, on a cheap VPS with no office and no money, has made me realize the value of thinking through your decisions. Don't forgo an office just because we don't have one. Think about what's best for you, your employees, and your business.

Start simple, incrementally improve, measure twice, and think about what you're doing.

Just like in shop class.

Thank you.

## Resources

Ruby Howdown 2008 Keynote

<http://gist.github.com/6443>

Startup Riot 2009 Keynote

<http://gist.github.com/67060>

DISCUSS: <http://railsmagazine.com/3/11>



# Interview with Carl Mercier

*interviewed by Rupak Ganguly on May 5th, 2009*

**Rupak:** Can you please tell us about your background, briefly, for the benefit of the readers?

**Carl:** I'm a self-taught programmer since the age of 7. I have a degree in Business Management and another one in Sound Engineering. I discovered Ruby (and Rails) shortly after the "Whoops" Rails video came out (<http://www.youtube.com/watch?v=Gzj723LkRJY>), so sometime in 2005. Although I've been programming pretty much forever, I consider myself more of an entrepreneur than a developer.

**Rupak:** How and when did you get involved in Ruby/Rails?

**Carl:** When I saw the Rails video, I thought its simplicity was astonishing, so I immediately became very interested in it. It was definitely a huge shift from C# and Visual Studio, which I was using at the time. I tried RoR on many small projects and was constantly frustrated by the lack of an IDE, debugging tools and cryptic error messages. So basically, I became frustrated, gave up and came back many times. The more I read about RoR, the more it started to make sense to me and I eventually completely stopped doing any Microsoft development to work with Ruby and Rails exclusively. But not without hiccups... <http://blog.carlmercier.com/2007/01/30/why-i-moved-from-ruby-on-rails-to-pythondjango-and-back/>. Since my return to Rails, life has been great. RoR is a lot mature now and it's easy to find help if you're stuck. I wouldn't want to work with anything else now.

**Rupak:** What drives you to work with Rails? What had drawn you towards this platform when you started and still catches your attention?

**Carl:** I love simplicity in general, so Rails is a great fit. You can get so much more done in so much less time with Rails, especially once you understand what happens "under the hood" (ie: behind the magic). Rails is really hard to beat for most applications. In our case (Defensio), we decided to use Merb for our API and Rails for our website. I think Merb is perfect for APIs since it has a pluggable architecture and less "bloated" overall.

**Rupak:** Would you hazard a prediction on the future of Rails? One year from now? Five?

**Carl:** I think enterprise adoption will start growing exponentially in the very near future. A lot of bigger consulting firms still believe Rails and Merb are "toys" and not ready for serious work. These firms are always at least 5 years behind (and wrong), so they'll likely catch up pretty soon.

**Rupak:** With Merb merging into Rails 3, what features from other platforms or frameworks would you like to be incorporated into Rails in future?

**Carl:** I think Merb has a much simpler code base. There's less meta-programming and magic going on, which I really like. That's probably my main wish for Rails 3. I would hate losing that. Another neat feature of Merb is the way dependencies are managed with `dependencies.rb`. I prefer it to Rails' dependency management and hope it makes the cut.

**Rupak:** Do you think Rails risks getting bloated with all this functionality? Or that it will lose its ease of use and magic as the price to pay for the newly found flexibility?

**Carl:** I personally think merging Merb and Rails is a big mistake, but I REALLY hope the team proves me wrong. The reason I think that is very simple: Merb and Rails attract different people for different reasons. Merb is lean, flexible, pluggable, fast and has an extremely easy to read/patch code base. The learning curve is a little steeper because everything is so configurable and flexible, but sometimes, that's what you want. On the other hand, Rails comes with great defaults and just works out of the box. No dependency problems or decisions about which ORM to use. I love that about Rails.

When we decided to rewrite our API with Merb, we wanted speed and flexibility. We felt that Rails had too many unnecessary bells and whistle for a simple API. For our website, I think it made a lot of sense to use Rails because of all the goodies that comes with it.

I find that Rails has been losing a lot of its simplicity lately, and that's scaring me. Learning Rails used to be extremely easy, just watch the infamous "Whoops" video in which DHH creates a simple blog in 5 minutes. Couldn't be easier than that.

Nowadays, creating the same simple blog using Rails' best practices is much more complex and confusing because it involves understanding REST principles, nested resources, migrations, etc. There's also many new terms floating around like Metal, Templates, Engines, Rack, etc that makes Rails more cryptic overall. This probably sounds very straight forward to most readers, but it is not to less experienced developers or Rails n00bs. I find that the learning curve has become steeper overall.

Merb focuses on flexibility, and Rails focuses on simplicity. I fear that the result of the merger will just be a compromise: Rails 3 will become more flexible (but less than Merb) and a lot more complex.

One thing I learned in my life is that you can't be everything to everyone, and I feel that Rails is trying to become exactly that. We'll see... Please prove me wrong!

**Rupak:** If you could give someone just starting out a tip, what would that be?

**Carl:** Do. Don't just read books, build small applications and put your newly acquired knowledge to work. Facing real-life problems and solving them is the best way to learn and become a better Ruby developer. (I'm talking to you, Remy! ;-)) Ruby.Reddit is a great place to learn new Ruby and Rails tricks, too.

**Rupak:** What would be one thing you wish would be there in Rails today?

**Carl:** While not directly related to Rails, I wish more gems were 1.9.1 compatible. The threading and speed of MRI is a bit pathetic. We sure can't wait to switch to 1.9.1!

**Rupak:** How was MeshU for you?

**Carl:** Had a blast! I'm particularly happy that my talk received such a great response. I'm surprised at the amount of email I'm getting about it. Definitely gives me a boost for my next talk!

**Carl Mercier** is a serial technology entrepreneur and developer from Montreal, Canada. His latest venture, Defensio, has attracted significant interest from worldwide blogger and developer communities and has exhibited superior filtering performance. Defensio was acquired by Websense Inc., a publicly-traded company in January 2009. Carl is now director of software development at Websense. Carl also founded Montreal on Rails, a developer community group that brings together local Ruby on Rails enthusiasts for monthly gatherings. Prior to founding Defensio, Carl started Adonis Technologies, a company that built advanced management software for the tourism & recreation industry. It was acquired in 2006. Carl holds degrees in business management and sound engineering. He also pretends to be able to beat just about anyone at poker. He blogs at <http://blog.carlmercier.com> and tweets at <http://twitter.com/cmercier>.



DISCUSS: <http://railsmagazine.com/3/12>

# PeepCode

s c r e e n c a s t s



PeepCode Screencasts are high quality tutorials that will get your team up to speed with **Git** source code control, help you learn **Ruby on Rails from Scratch**, introduce your designers to the **Command Line**, take you in-depth with **RSpec**, teach you **Objective-C** and **MacRuby**, not to mention **XMPP**, **CouchDB**, **REST**, **Capistrano** and **Phusion Passenger™** deployment, **Javascript**, **Hamli & Sass**, and other topics you need to know about.

<http://peepcode.com>

Only \$149 for a full year, or \$9 for a single screencast.



# Interview with Ilya Grigorik

*interviewed by Rupak Ganguly on May 8th, 2009*

**Rupak:** Can you please tell us about your background, briefly, for the benefit of the readers?

**Ilya:** I'm the founder and chief Ruby wrangler at AideRSS, a startup based in Waterloo, Canada, where we're building a collection of services for both the consumers and publishers of RSS content. More specifically, PostRank which is our core technology is an algorithm we've developed to rank any online content based on the overall 'audience engagement' – how many people have bookmarked a story, dugg it, shared it on twitter, etc. We gather all the social engagement data in real-time and make it available in a form of a filter (give me only the best stories from this feed), or as real-time data feed for publishers.



**Ilya Grigorik** is the founder of AideRSS – a real-time social media engagement monitoring and analytics platform.

He has been wrangling with Ruby and cloud computing for over four years, trying to make sense of it all. In the process, he has contributed to numerous open source projects, blogged about his discoveries (blog: [www.igvita.com](http://www.igvita.com), twitter: @igrigorik) and as of late, has been an active speaker in the Ruby, Social Media and Cloud Computing communities.

Previous to AideRSS I was running my own web-hosting company, doing some freelance work on the side, and running a couple of hobby sites while I was attending University of Waterloo. One thing led to another, the idea for AideRSS was born, we managed to find a financial partner, and here we are!

**Rupak:** How and when did you get involved in Ruby/Rails?

**Ilya:** I got into Rails around mid-2005 while working on a personal project. Up to that point, I've been working primarily with PHP (after migrating from Perl), but given all the press around RoR at the time I decided to give it a try. Haven't looked back since.

As with many Ruby/RoR developers I joined the community because of RoR, but stayed because of Ruby. At the time I was also doing some work in Python, but the elegance of Ruby really appealed to me and before I knew it I was converting all of my Perl/PHP code into Ruby just for the sake of getting hands-on practice with Ruby.

**Rupak:** What drives you to work with Rails? What had drawn you towards this platform when you started and still catches your attention?

**Ilya:** There are a lot of things that Rails got right, right out of the gate. Convention over configuration creates a bit of confusion when you're getting started, but ultimately it is a liberating experience – especially when the project grows larger and you have multiple people working on it. Likewise, database migrations in ActiveRecord! To this day, most other frameworks are still playing catch up.

In fact, we can go on for hours: clean MVC model, the fact that testing is easy and part of the regular workflow, an extensible plugin platform, etc! And perhaps most importantly, the community around it.

Technology is what gave Rails the initial push, the community around it is what made it such a success.

**Rupak:** Would you hazard a prediction on the future of Rails? One year from now? Five?

**Ilya:** Sunny. I'm continuously amazed at the new features being integrated into every release of Rails. Our community still attracts much of the thought leadership in testing, deployment, and library support and perhaps even more importantly, we are open-minded about better or alternative solutions. The merge of Merb and Rails is a great example of this.

As a framework, Rails is definitely maturing and it's exciting to see all the companies (EngineYard, Heroku, New Relic, etc.) being created around the tooling, deployment, and management of the infrastructure.

Trend for the next year? I think we're going to see a lot more enterprise deployments of Rails, which in turn opens up an entirely new ecosystem: support, consultants, tooling, and so on.

**Rupak:** With Merb merging into Rails 3, what features from other platforms or frameworks would you like to be incorporated into Rails in future?

**Ilya:** Django's debug bar (already underway as a Rack middleware) and administration panel (available as a merb plugin) is something I would love to see in the core of Rails at some point. More broadly, I'm looking forward to having a more modular framework as a whole.

On one hand I'm a big fan of the principle of convention on which Rails was founded, on the other, the modularity of Merb is a must have for many developers. Merging these two concepts will go a long way towards meeting the needs of a



wider audience and growing out of the MySQL/LAMP stack assumption.

**Rupak:** Do you think Rails risks getting bloated with all this functionality? Or that it will lose its ease of use and magic as the price to pay for the newly found flexibility?

**Ilya:** I think the Rails core team has a good philosophy around this. One of the motivations behind merging Merb and Rails is to cherry pick the best concepts from each, and one of the things Merb got right was the concept of a modular framework. Moving forward, Rails will see more of that, which means you will be able to pull in libraries or entire functional components into the framework and customize it to your needs without 'bloating the framework' itself.

Of course, Rails will still come preconfigured with sensible defaults right out of the gate, and I'm guessing the vast majority will stick with the defaults, but if you really need a different database adapter, you'll be able to make it so!

**Rupak:** If you could give someone just starting out a tip, what would that be?

**Ilya:** Watch the 'build a blog in 5 minutes video' and follow it along! Hands-on experience is the best way to learn, and it will also let you experiment and discover the beauty of both Rails and Ruby firsthand. Beyond that, a good book always helps, so visit your local bookstore, and then do a Google search for a local Ruby / Rails user group or a meetup. I guarantee it, you'll be pleasantly surprised at how helpful other Rubyists will be as you begin your foray into the wonderful world of Ruby!

**Rupak:** What would be one thing you wish would be there in Rails today?

**Ilya:** An easier deployment model and more ubiquitous Rails support amongst hosting providers. Phusion Passenger is a great step forward as it eliminates an entire class of infrastructure (reverse proxies and distinct app servers), but I'm still hoping for more alternatives moving forward. Heroku and other hosting providers are building their own 'one-click deploy' models, which is great to see, but I'm also keeping my fingers crossed for JRuby + Glassfish as an easy alternative – it would mean one click deploys on any JVM stack.

**Rupak:** How was MeshU for you?

**Ilya:** It was a fantastic event, I really enjoyed both Mesh and MeshU. If you're in Toronto/GTA area and haven't attended, definitely check it out in 2010. What I enjoyed most was the relatively small size of the event, which meant that you could easily chat with the speakers about their talks and go in depth on any given topic. Looking forward to the next one already!

DISCUSS: <http://railsmagazine.com/3/13>

## Artist Profile

Illustration for this issue was kindly provided by **David Heinemeier Hansson**.

David was born in Copenhagen, Denmark and moved to Chicago, US a few years ago. Equipped with a Canon EOS 5D Mark II, David is practicing photography at home and in his travels.

David is best known as the creator of Ruby on Rails.

Photo gallery: <http://www.flickr.com/photos/46457493@N00/>

Twitter: <http://twitter.com/dhh>

Blog: <http://www.loudthinking.com/>

Wikipedia: [http://en.wikipedia.org/wiki/David\\_Heinemeier\\_Hansson](http://en.wikipedia.org/wiki/David_Heinemeier_Hansson)



Front cover: "Munching strawberry" by David Heinemeier Hansson

Back cover: "Howling in the Park" by Huw Morgan

DISCUSS: <http://railsmagazine.com/3/14>



# Interview with Ryan Singer

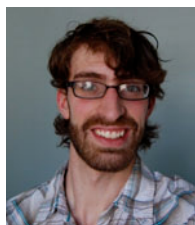
interviewed by Rupak Ganguly on May 5th, 2009

**Rupak:** Can you please tell us about your background, briefly, for the benefit of the readers? How and when did you get involved in Ruby/Rails?

**Ryan:** I started with web design when I was a teenager and freelanced until I joined 37signals in 2003. At that time we were a UI design consultancy, mainly doing redesigns of existing websites. Shortly after I came on board we started designing Basecamp and connected with DHH. After Basecamp launched in 2004 and Rails was extracted, we found ourselves focused on writing applications instead of consulting for clients. Parallel to the company's transition, I also found myself becoming more interested in software and found my role as UI designer going beyond "web design" to software design.

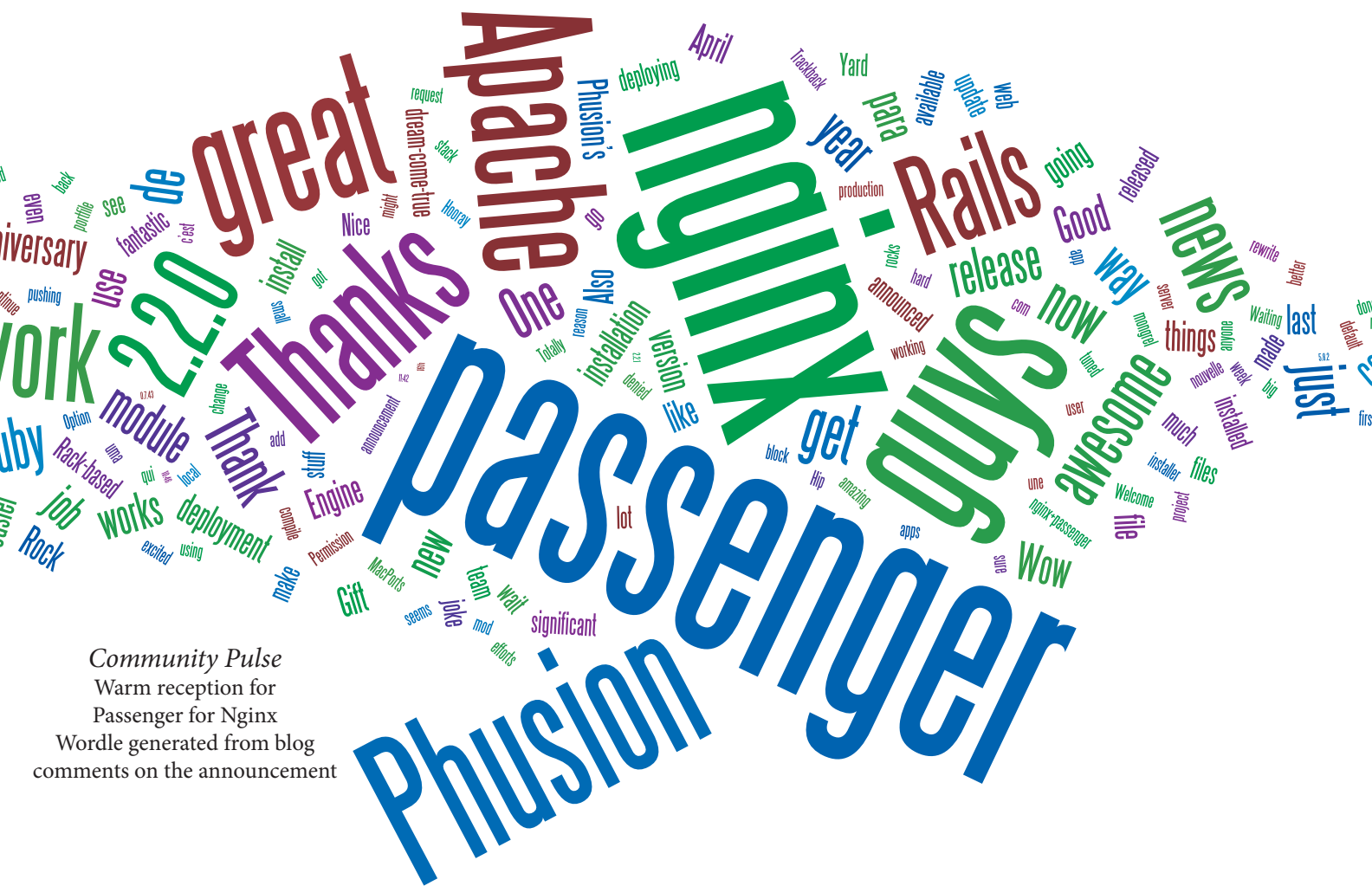
**Rupak:** What drives you to work with Rails? What had drawn you towards this platform when you started and still catches your attention?

**Ryan:** The key thing about Rails is it lets me work at the right level of abstraction. The amount of implementation I have to think about is small enough that I feel like I'm touching the model directly and the views are 90% HTML. Also



**Ryan Singer** is an interface designer and product manager at 37signals. Ryan joined 37signals in 2003. Within a year the company transitioned from a design consultancy to a software company with the release of Basecamp. Ryan's interface designs have since set a standard for web application usability and clarity. He lives in Chicago with his wife and french bulldog.

the MVC pattern is a major support for our work at 37signals as a UI-focused company. Designing UI in a stubbed Rails app is so much better than writing independent mock HTML screens because I can see how a design behaves with real data as I build it. Working with partials and helpers also means that my templates do a lot without sacrificing readability. I'm a huge fan of Eric Evans' Domain-Driven Design, and I feel like Ruby and Rails are an ideal environment for working with those patterns.



Community Pulse

Warm reception for  
Passenger for Nginx

Wordle generated from blog  
comments on the announcement

# Rails Magazine Team

**Rupak:** Would you hazard a prediction on the future of Rails? One year from now? Five?

**Ryan:** I can only hope that as time goes by people will appreciate the rare opportunities Rails gives them. Rails makes it easier for developers and designers to collaborate, so I hope we see more UI designers getting involved in software design. I'd like to see more discussion happening about the relation between modeling and UI, and perhaps more conversation about view code conventions. Most of us on the UI side are relying on patterns we inherited from the standards/CSS movement, which are a good basis but they're also not enough. I'm doing my best to share the patterns I've found as the opportunities present themselves.

**Rupak:** With Merb merging into Rails 3, what features from other platforms or frameworks would you like to be incorporated into Rails in future?

**Ryan:** That's outside my knowledge probably.

**Rupak:** Do you think Rails risks getting bloated with all this functionality? Or that it will lose its ease of use and magic as the price to pay for the newly found flexibility?

**Ryan:** That's outside my knowledge probably.

**Rupak:** If you could give someone just starting out a tip, what would that be?

**Ryan:** I'd advise anyone doing UI for Rails apps to take responsibility for the view directories. Share the source code, own those directories, and don't lean on programmers to implement your designs. From there it's a small step to find out where those instance variables are coming from, what a model is, or how helpers can help you out. UI designers don't have to become programmers, but we should know what MVC is and where we belong in it.

**Rupak:** What would be one thing you wish would be there in Rails today?

**Ryan:** I'm quite happy with it. I may not be technical enough to answer this question.

**Rupak:** How was MeshU for you?

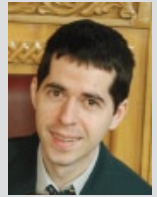
**Ryan:** MeshU was great. Everyone had their lights on and seemed excited to be there. I met a lot of interesting folks between sessions and generally found it very inspiring.

DISCUSS: <http://railsmagazine.com/3/15>

**Olimpiu Metiu**

Editor-in-chief

<http://railsmagazine.com/authors/1>



**John Yerhot**

Editor

<http://railsmagazine.com/authors/2>



**Khaled al Habache**

Editor

<http://railsmagazine.com/authors/4>



**Rupak Ganguly**

Editor

<http://railsmagazine.com/authors/13>



**Mark Coates**

Editor

<http://railsmagazine.com/authors/14>



**Starr Horne**

Editor

<http://railsmagazine.com/authors/15>



**Bob Martens**

Editor

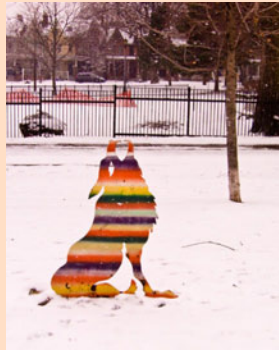
<http://railsmagazine.com/authors/16>



## Call for Papers

*Top 10 Reasons to Publish in Rails Magazine*

1. **Gain recognition – differentiate and establish yourself as a Rails expert and published author.**
2. Showcase your skills. Find new clients. Drive traffic to your blog or business.
3. Gain karma points for sharing your knowledge with the Ruby on Rails community.
4. Get your message out. Find contributors for your projects.
5. Get the Rails Magazine Author badge on your site.
6. You recognize a good opportunity when you see it. Joining a magazine's editorial staff is easier in the early stages of the publication.
7. Reach a large pool of influencers and Rails-savvy developers (for recruiting, educating, product promotion etc).
8. See your work beautifully laid out in a professional magazine.
9. You like the idea of a free Rails magazine and would like us to succeed.
10. Have fun and amaze your friends by living a secret life as a magazine columnist :-)



## Call for Artists

*Get Noticed*

Are you a designer, illustrator or photographer?

Do you have an artist friend or colleague?

Would you like to see your art featured in Rails Magazine?

Just send us a note with a link to your proposed portfolio. Between 10 and 20 images will be needed to illustrate a full issue.

## Join Us on Facebook

<http://www.facebook.com/pages/Rails-Magazine/23044874683>

Follow Rails Magazine on Facebook and gain access to exclusive content or magazine related news. From exclusive videos to sneak previews of upcoming articles!

Help spread the word about Rails Magazine!

## Contact Us

*Get Involved*

Contact form: <http://railsmagazine.com/contact>

Email: [editor@railsmagazine.com](mailto:editor@railsmagazine.com)

Twitter: <http://twitter.com/railsmagazine>

Spread the word: <http://railsmagazine.com/share>

## Sponsor and Advertise

*Connect with your audience and promote your brand.*

Rails Magazine advertising serves a higher purpose beyond just raising revenue. We want to help Ruby on Rails related businesses succeed by connecting them with customers.

We also want members of the Rails community to be informed of relevant services.

## Visit Us

<http://RailsMagazine.com>

Subscribe to get Rails Magazine delivered to your mailbox

- Free
- Immediate delivery
- Environment-friendly

## Take our Survey

*Shape Rails Magazine*

Please take a moment to complete our survey:

<http://survey.railsmagazine.com/>

The survey is anonymous, takes about 5 minutes to complete and your participation will help the magazine in the long run and influence its direction.

